Frame, *Rods* and *Beads* of the Edge Computing *abacus*

A Thesis
Presented to
The Academic Faculty

by

Ketan Bhardwaj

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

School of Computer Science
Georgia Institute of Technology
December 2016

Frame, *Rods* and *Beads* of the Edge Computing *abacus*

Approved by:

Professor Ada Gavrilovska, Committee Chair
School of Computer Science
Georgia Institute of Technology

Professor Karsten Schwan, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Ling Liu
School of Computer Science
Georgia Institute of Technology

Professor Dilma Da Silva
Department of Computer Science
and Engineering
Texas A&M University

Professor Mostafa Ammar
School of Computer Science
Georgia Institute of Technology

Professor Ellen Zegura
School of Computer Science
Georgia Institute of Technology

Professor Taesoo Kim
School of Computer Science
Georgia Institute of Technology

Date Approved: 11/11/2016

To my wife - Sakshi Sharma*, son* - Aadiv Bhardwaj *and my Parents.*

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude and thank my advisers Prof. Karsten Schwan and Prof. Ada Gavrilovska without whom this dissertation would not have been possible. It has been an honor to be their Ph.D. student.

From their support in terms of allowing me to defer start of my Ph.D. even before I joined the group, to allowing and supporting me to pursue very high risk research projects, to bearing my ridiculously long and random *idea* emails, to keeping me focused when I wandered, and finally, their contributions in terms of time, ideas, and research directions making my Ph.D. experience the most intellectually stimulating and enjoyable time in my career so far. They helped me understand the importance of articulation and presentation of ideas. Their support, motivation and guidance has contributed to this dissertation greatly.

I would like to thank the other members of my dissertation committee, Prof. Ling Liu, Prof. Mostafa Ammar, Prof. Ellen Zegura, Prof. Dilma Da Silva and Prof. Taesoo Kim for serving on my dissertation committee and for their insightful comments and suggestions on my research.

My colleagues and mentors during my internships Dilma Da Silva (Qualcomm), Stephen Ludin, Moritz Stiener and Martin Flack (Akamai). They have had significant impact on shaping up my thinking that led to ideas in this dissertation.

I interacted with a number of people during my time at Georgia Tech which made it enjoyable and intellectually stimulating at the same time. A mention to friends Dipanjan Sengupta, Abhinav Narayan, Minsung Jang, Sudarsun Kannan, and Bharath Srinivasan. A special mention to my friends and collaborators Pragya Agarwal and Ming-Wei Shih who worked with me on some of the projects thereby actively contributing to this dissertation.

Susie McClain, our group admin, who has been extremely helpful in taking care of our travels, day to day lab needs, etc. deserves a special approbation.

Finally, I would like to thank my family: my wife for her support, my son for being my stress buster when things got hectic and my parents for their encouragement over the years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Going beyond point solutions, the vision of edge computing is to assimilate itself in the existing computing ecosystem to enable web services to deploy their edge functions (EF) in a multi-tenant edge infrastructure. However, there are critical challenges in making that vision possible. This dissertation addresses three such critical challenges:

1. Demonstration of benefits of edge functions for highly dynamic and large scale Android app ecosystem: (i) AppFlux and AppSachets, to relieve bandwidth pressure due to existing app delivery mechanisms, (ii) Ephemeral apps and app slices that rethink the app delivery for emerging app usage models, to highlight that the edge computing can enable transformational changes in the computing landscape beyond just latency and bandwidth optimizations.

2. Design and implementation of AirBox – a secure, lightweight and flexible edge function platform needed by web services to deploy and manage their EFs on edge computing nodes on-demand. AirBox is based on a detailed experimental design space exploration for system level mechanisms that are suitable for an edge function platform to address the technical challenges associated with provisioning, management and EF security. AirBox leverages state-of-the-art hardware-assisted and OS-agnostic security features, such as Intel SGX, to prescribe a reference design of a secure EF.

3. Finally, a solution to the most critical issue of enabling edge functions while preserving end to end security guarantees. Today, when most web services are delivered over encrypted traffic, it is impossible for edge functions to provide meaningful functionalities without compromising security or obviating performance benefits of EFs. Secure protocol extensions (SPX) can efficiently maintain the proposed End-to-Edge-to-End (E3) security semantics. Using SPX, we accomplish the seemingly impossible task of allowing edge functions to operate on encrypted traffic transmitted over secure protocols, while ensuring their security semantics, and continuing to provide the benefits of edge computing.

CHAPTER I

THE CASE FOR EDGE FUNCTIONS

The trends in computing systems can be broadly characterized by oscillations between centralized (e.g., main frames, the cloud) and distributed approaches (e.g., peer-to-peer). The current direction of oscillation towards a distributed approach is being referred to as Edge Computing. This is driven by emerging applications enabled by powerful end-user devices and 5G technologies which pose demands for reduced access latencies to web services [44] and dramatic increase in the backhaul network capacity [13, 41].

1.1 Edge Computing

Edge computing—the use of computational resources closer to end devices, at the edge of network—is becoming an attractive approach to addressing the latency and backhaul bandwidth demands of emerging applications. This is evident from various solutions targeting different types of applications, such as video caches [34], IoT gateways [33, 15], as well as commoditization of access network components [29, 74]. To put this in context, all major mobile network operators (e.g., AT&T, Verizon) are looking to allow third parties (e.g., Netflix, Facebook) to deploy edge functions in the compute infrastructure they intend to provision at their network's edge and to create new revenue generating streams.

Going beyond point solutions, Figure 1 shows the vision of edge computing, where web services deploy their *edge functions* in a multi-tenant infrastructure, as proposed by researchers [59, 20, 137] and industry initiatives [21, 38].

1.1.1 Motivating Problems

On a high level, edge computing is driven by the inability of the cloud to provide access latencies close to 1 millisecond due to physical constraints, i.e., the speed of light which mandates deployment of a data center every 300 kilometres to achieve it. At the same time, creating and operating distributed

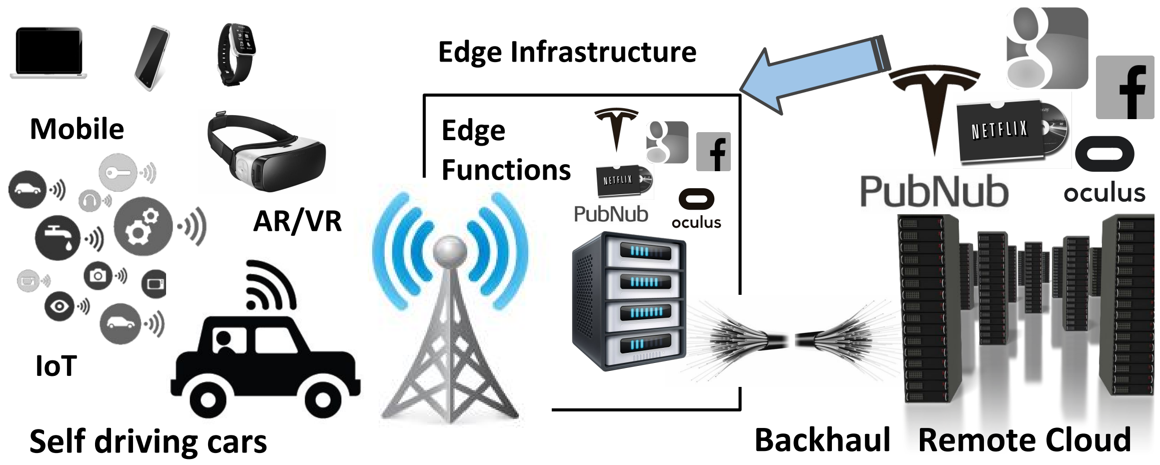


Figure 1: Landscape of edge computing.

deployments of large data centres is an unreasonable proposition. The emergence of Edge computing is a result of trying to find the right trade-off to distribute computational capability to devise solutions that address the fundamental problems in the current computing ecosystem in form of:

- **Unpredictable high latency:** Mobile-Cloud applications often face, and hence are designed to handle, unpredictable access latencies to the remote cloud. But, even in their best cases, the achieved latencies are still higher than the sub-millisecond level demanded by next generation applications, e.g., tactile internet, driver-less cars, augmented reality, etc.
- **Prohibitive bandwidth usage:** Increasing use of back-haul bandwidth by web services will exacerbate in the future due to the sheer number of devices that would need Internet connectivity, including the ones proposed as part of Internet of Things (IoT) scenarios. End users have to face the burden of those increasing costs. The costs increase either directly, in form of increase consumption in their data plan costs, or indirectly, which is then transferred to end users.

1.1.2 Technology Drivers

The technology evolution that is driving the edge computing include the following:

- **The last mile problem:** Access to remote clouds is fundamentally limited by the speed of light which mandates a data centre every 300 kilometres to achieve sub millisecond access latencies required by emerging use-cases such as tactile Internet, driver-less cars, augmented reality, etc.

Further, important to note here is that traditional CDNs cannot mitigate redundancy of traffic or latency in the last mile simply because they operate on its edge facing ISPs as opposed to edge which operate on the client side. In addition, the ISPs often opt to run their own CDNs to exploit redundancy in traffic and/or reduce latency passing through their pipes. But, they remain ineffective for non-cachable content flagged using HTTP headers, cookies, etc., due to the pay-per-download nature and/or issues related to intellectual property protection, or for content served using encryption. The pain point in the content delivery is the congestion and latency in the last mile of the Internet. Edge computing nodes operating past ISPs, near end users, can potentially effectively leverage redundancy and mitigate the latency at the same time. These potential benefits are one of the factors driving the interest edge computing.

- **Availability of cheap processing and storage:** Increasingly powerful platforms are being deployed at the edge in network elements like wireless routers, cellular towers, aggregation points, or as standalone severs at strategic locations, etc., due to the reduced costs of compute resources. Storage benefits from similar technology trends in terms of reduced cost of storage capacity. The potential benefits of replicating the success of cloud computing at the edge is another factor driving the edge computing.

- **Increasingly Fast Wireless Networks:** The theoretical access data rates of on-device flash memory [1] can be an order of magnitude lower than the wireless network (cellular and WiFi (802.11x)). This indicates a strong potential performance benefit from accessing functionalities over the local wireless network vs. even a device's local flash device, let alone remote cloud. While such ideal numbers cannot be taken at face value given the practical issues in achieving those data rates, with new adaptive methods for wireless communications [122, 112], however, demonstrate that such practical issues can be overcome. This is another factor driving edge computing to solve the latency problem faced by mobile-cloud computing.

1.1.3 Policy Drivers

The policy landscape driving the edge computing can be summarized by the following two policies:

[1]ISSCC Trends vs. JEDEC Mobile Memory Technology Road map.

- **Net Neutrality:** On one hand, the recent policy initiatives geared towards ensuring the continued social and economic benefits of an open Internet eco-system have proposed that Internet service providers including mobile network operators should enable access to all content and applications regardless of the source, and without favoring or blocking particular products or websites. In effect, this has constrained the ability of Internet service providers or mobile network operators to differentiate their services solely based on their deployed network and hence, limits the avenues of generating additional revenue that drives the deployment of newer, better and faster network. Their search for alternatives which leverage their strengths, i.e., the already deployed distributed network elements which can be augmented with computational capability, leads to edge computing.

- **Cyber-security and Privacy:** On the other hand, cyber-security policies and privacy policies are professing the use of end-to-end encryption in web access due to obvious problems related to fraud, IP theft and surveillance concerns. This trend can make it close to impossible for Internet service providers or mobile network operators to optimize the use of their communication infrastructure. Coupled with the humongous amount of traffic expected from emerging applications, this poses significant financial pressure on them to ready their communication infrastructure. This is another factor where policies' implications are driving the use of computational elements near end users to manage the traffic and hence, leading to edge computing.

- **New revenue generation:** Today, most revenue in the Internet ecosystem is generated by Internet service provider companies like Google, Facebook, Netflix, etc., by infrastructure companies like Amazon, Microsoft, etc., or by application developers, e.g., mobile app developers. Despite being a critical component of the Internet ecosystem, mobile networks like AT&T, Verizon etc., are at the brink of becoming a commodity entity owing to recent policy initiatives on net-neutrality and privacy, making it impossible for them to create new revenue streams from their network. So, all major operators are looking for new ways to monetize their networks and create new revenue opportunities. Edge computing can provide them with literally an edge to differentiate themselves and provide valuable low latency services to end users and infrastructure services to Internet companies. This is the major factor driving interest in edge computing.

1.2 Edge Functions

Today, edge computing remains loosely defined to describe geographical dispersed computational infrastructure element outside of data center based cloud which is only one wireless hop away from end users. Heterogeneity in their capabilities, lack of deployment models or system support or for that matter concrete use cases demonstrating real world benefit make it a non-trivial task to argue about uses of those platforms. Broadly, Edge Computing is being explored in the following two major research directions:

- **Client driven cyber-foraging:** One direction in research suggests the use of *client-driven* cyber-foraging to utilize edge clouds. Examples of which include offloading system like MAUI [69], Clone-Cloud [68], Comet [82] to overcome resource constraints of mobile devices to reduce delay in user perceived response. However, these approaches are effective only when the delay in response is compute bound. Other proposals like NOMAD [127] take a client driven approach to minimize the user perceived maximum access latencies.

  Despite its promising benefits, client-based offloading is faced with many difficult technical challenges arising out of the diversity in the end user device space – from supporting numerous types of devices, their diverse OSes, and numerous apps running on them, to accurate code profiling, and gauging optimal offload conditions, which often requires continuous monitoring of network conditions on resource constrained end user devices. At the same time, back-end driven approaches are inherently designed to handle different types of devices. They also have access to practically unlimited computational capability at their disposal, powered by the cloud, to accurately characterize access to their service usage with respect to users, location, and time, which they can use to appropriately onload services at edge clouds or cloudlets.

- **Backend-driven cyber-foraging:** Another promising direction, the one proposed in this thesis as well and depicted in Figure 22, is *backend-driven* cyber-foraging that proposes to run appropriate beneficial services (e.g., caching content, accelerating relevant traffic, etc.) or even including some backend functionality (e.g., buffering notifications, aggregating redundant traffic, etc.) near end-users, with a goal of minimizing user perceived latency and consolidating the use of bandwidth.

Examples of such backend-driven approach and has been demonstrated for different services, including AppFlux [58], that allow end users to instantly access app updates via edge-cloud-based app streaming at speeds 2x faster compared to content delivery networks (CDN), cognitive assistance application based on Cloudlets [86], that show the feasibility of moving functionality between cloudlet servers and the cloud. Additionally, backend-driven onloading unlocks the opportunity to consolidate bandwidth usage at the edge, by employing service-specific logic to reduce traffic over the Internet, e.g., via filtering, compression, or caching in edge services. This can reduce the cost incurred by end users in the form of reduced data charges, as well as by backend services in terms of reduced use of network bandwidth. For instance, for app updates as a service, AppSachet [55] demonstrates opportunities to reduce traffic in the last mile by up to 83%.

1.2.1 Definition

In this thesis, we argue that a back-end driven cyber-foraging approach is more pragmatic than client driven cyber-foraging. Accordingly, we coin the the term edge function as defined below.

Edge Function (EF): *Any third party service deployed on edge infrastructure that interacts with end client requests on behalf of a back-end service deployed in remote clouds.*

Typically, EF is functionality implemented over networking layer or layer 4 because it requires or uses back-end service's specific logic or semantic information that is not available at the network layer. A simple example of this difference is network level caching which is an example of network function vs. replication of web service content among geographically distributed data centers which inherently requires a application specific knowledge to carry it out effectively. In the same vein, we call the underlying software stack as Edge Function Platform or EFP that provides system support for deployment, execution and security of edge functions.

1.2.2 Benefits

Fundamentally, following benefits of using edge functions are derived from their shorter distance to end users and a vantage point beyond the last mile:

- **Latency reduction:** EFs can make possible access latencies of less than 1 ms observed by applications on end user devices, a feat that is impossible to achieve with remote cloud based services due to network distance between end user device and remote clouds (including those set up by content delivery networks) and physical constraints imposed by the speed of light.
- **Bandwidth consolidation:** EFs can open opportunity to deploy functionalities such as aggregation, filtering or at the very least, buffering to provide flexibility in scheduling the use of bandwidth to non-peak hours. Thus, providing consolidation in the use of bandwidth by end user, the upcoming IoT devices and back end services.

1.2.3 Replicating cloud at the edge

Despite these seemingly obvious benefits and substantial momentum from industry towards edge computing, one can argue that edge computing is simply application of the solutions developed in the cloud computing space to edge infrastructure. However, in this thesis, we argue that this is not the case. Although, there are obvious benefits of deploying supporting functionalities e.g., proxy, load balancers etc., as edge functions, using same mechanisms and solutions developed as part of cloud computing but there are assumptions that do not hold, additional constraints and opportunities in edge computing that do not manifest themselves in the cloud space. We hypothesize that there are critical barriers for edge functions to assimilate in to the mainstream computing ecosystem that will result in harnessing the full potential of edge computing.

1.3 Thesis Statement

To harness full potential of edge computing, edge functions must be assimilated into the mainstream computing ecosystem through concrete characterization and demonstration of the edge function benefits, design and development of appropriate system support to dynamically and securely deploy edge functions, and by addressing the most critical challenge of making it possible for edge functions to operate on encrypted traffic without compromising end-to-end security semantics or performance.

1.4 Contributions

The key contribution of our research is a set of technologies that addresses the aforementioned challenges. Specifically, to validate the thesis we make the following contributions:

- Introduces following abstract notions relevant to Edge computing.

    1. *Edge functions.* as a fundamental construct that delivers the benefits of Edge Computing to mainstream computing applications.
    2. E^3 *security semantics.* A fundamentally new type of security semantics without which the whole edge computing paradigm can potentially be rendered useless.

- Demonstration of edge functions designed to address pain points of Android app eco-system:

    1. *AppFlux.*, that allow end users to instantly access app updates via app streaming as an edge function at speeds 2x faster compared to content delivery networks (CDN) and reducing traffic due to app installs and updates by 70%, without requiring any modifications to the existing apps [58].
    2. *AppSachet.* demonstrates opportunities and proposes intelligent caching policies to further reduce app traffic in the last mile by 13% taking a total reduction to 83% [55] .
    3. *AppInstant.* that uses the proposed *ephemeral app model* and the notion of *app slices* powered by edge functions to re-envision the app delivery model. The proposed model reduces barriers in app discovery and allows providing fine grained functionality - app slices - to end users to maintain relevancy of app eco-system with emerging ways of web service consumption e.g., messaging apps.

- **AirBox.** A software platform – *frame* – that enables flexible, fast and scalable edge function onloading while ensuring security for its execution and stored state – *rods* – evaluated using the proposed generic edge functions – *beads*. AirBox allows web services to deploy their EFs just in time while providing management capabilities to their EFs on edge computing nodes. The research contributes to understand the trade-offs in using different system level mechanisms for developing an edge function platform. The experimental design space exploration focuses on technical aspects crucial for edge functions i.e., provisioning, management and security. AirBox leverages state or art hardware assisted OS agnostic security features i.e., Intel SGX to prescribe a reference design of a secure EF.

- **Secure protocol extensions (SPX).** A solution to the most critical problem that could render the whole edge computing paradigm useless. Today, when most web services are delivered over end-to-end encrypted traffic, using SPX we demonstrate how to accomplish the seemingly impossible task of allowing edge functions to operate on encrypted traffic, while ensuring that end-to-end security semantics of secure communication protocols. Secure protocol extensions can efficiently maintain the proposed End-to-Edge-to-End (E^3) security semantics.

1.5 Dissertation Structure

The rest of this dissertation is structured as follows.

Chapter 2 motivate use of edge functions by describing and quantifying the concrete pain points in app delivery along with relevant details about Android app ecosystem.

Chapter 3 describes the edge function based solutions in details demonstrated to address those pain points by co-designing them with novel Mobile OS support.

Chapter 4 outlines the technical challenges to translate similar benefits for arbitrary web services using generic edge functions.

Chapter 5 describes the exploration of system level technologies to ensure fast, flexible and secure onloading of edge functions. Then, it describes AirBox - a generic edge function platform.

Chapter 6 describes a protocol level solution to enable use of edge functions operation over encrypted traffic. It describes how to achieve the appropriate E^3 - End-to-edge-to-end security semantics in existing secure protocols.

Each of the above chapters also include a detailed performance evaluation of each of the solutions validating their effectiveness when compared to state-of-the-art competing solutions.

Chapter 7 discusses the salient research relevant to the work presented in this thesis.

Chapter 8 describes some relevant problems that were identified but not addressed completely at the time of preparations this thesis and which can be considered future directions of this research.

Finally, we close with concluding remarks in Chapter 9.

CHAPTER II

THE ROLE OF EDGE COMPUTING FOR APP DELIVERY

In order to demonstrate real world benefits of using edge functions, we chose to develop edge functions that address concrete problems in app delivery for mobile app ecosystem. The rationale behind choosing Android app ecosystem is its popularity, scale, highly dynamic nature and openness that allowed us to develop real end to end systems. In this chapter, we describe the pain ponts of Android app eco-system §2.2 and relevant details about Android app ecosystem to formulate practical edge function based solutions.

2.1 Introduction

The number of active smart devices worldwide has surpassed 1 billion, and is forecast to cross the 3 billion mark by 2018. Coupled with these tremendous increases in the number of active devices is an explosion in the apps available to end users, with roughly 80 billion app downloads reported for 2013, set to reach a staggering 200 billion by 2017 [1]. App installs and more so, app updates affect users both directly, by potentially consuming costly bytes in their data plans and indirectly, by congesting *the last mile* of the Internet known to be a bottleneck for delivered service quality [28]. The current app delivery mechanisms, where apps are delivered directly from app stores ignores such issues. In the fast paced app market, a conservative approach to updates lead to other issues like retaining app users' interests, delay in security and performance fixes. The consequences are undesirable increases in data plan costs for end users and increased congestion in the last mile, even in the presence of Content Delivery Networks (CDNs).

Moreover, even if the last mile congestion can be solved by using edge functions presented, there are more difficult problems faced by Android app ecosystem. One critical problem is App Discovery. Concretely, The number of apps available in the Google play store neared 1.5 million by the end of 2014. There app ecosystem has gained substantial momentum with new apps beign

[1]Data Source: http://mobithinking.com/

introduced at a much faster pace than they are consumed by end users. In fact, only 15% of total available native apps are downloaded more than 5,000 times[2]. The app discovery gap is likely to widen with the impending explosion of new apps prompted by the wearables, on-demand interactions in Internet-of-Things, situation-specific apps, etc. Existing mechanisms do not reduce the burden to try new apps for end users

Furthermore recently, how end user interacts with web services is evolving e.g., the ubiquitous use of messaging apps, end users' desire for more efficient, easy to find online services, and advancements in artificial intelligence, are posing chatbots based on messaging apps as a main vehicles of contextual service delivery and consumption [76, 81, 121]. Chatbots provide human like responses over a text based interface to offer personal assistant like services for shopping, transportation, personalized news, etc. Chatbots are deemed to pave the way for the next disruption in mobile systems, often considered as *the end* of the apps era. However, it would requiring mobilization of a new developer community centered around chatbot based apps and yet another refactoring of web services similar to what happened for apps, which is expensive and difficult to do. Existing mechanisms in app eco-system are not sufficient to ensure relevance of apps as we go forward.

In this chapter, we also outline solutions developed as a response to these problems. We then, outline high level benefits expected from such edge functions and practical constraints in designing §2.3 which are crucial for their deployability. Then, we present the measurements §2.4.1 and studies §2.2.3, §2.2.2 carried out to quantify those painpoints. We end this chapter with required Android background §2.4.4 from which we elicit design considerations for the presented solutions. In next chapter, we delve deeper in to details about the developed solutions.

2.2 Pain points of Android app ecosystem

The work carried out as part of this thesis addresses the following concrete painpoints for app ecosystem:

[2]Data source: http://www.appbrain.com/stats/

2.2.1 Supporting frequent app updates without congesting the last mile

App installs and more so, app updates, which our studies have found to exceed installs by more than a factor of 3x (see Figure 2(a)), affect users both directly, by potentially consuming costly bytes in their data plans and indirectly, by congesting *the last mile* of the Internet known to be a bottleneck for delivered service quality [28]. The current app delivery model, where apps are delivered directly from app stores, ignores such issues, which suggests that a conservative approach to rolling out app updates would lessen burdens on end users. Unfortunately, the fast paced nature of today's app market mandates frequent app updates to retain users, by offering new features, changing app aesthetics, enhancing usability, dealing with bugs, improving security and performance.

In response we developed *AppFlux* and *AppSachets* for the Android app ecosystem as an additional path for app delivery. It offers an alternative way to handling the delivery of apps and their updates via app streaming.

AppFlux comprises (i) novel system support in Android for *streaming* apps, which can eliminate the need for explicit updates and enable *just-in-time* delivery of usage based app updates and (ii) edge function that cache apps and app updates on behalf of end users. It reduce last mile traffic volume by leveraging (i) their better connectivity to app stores and end user devices, and (ii) the redundancy inherent in app traffic. Gains are derived from the fact that edge function can update apps frequently, without requiring any action from a mobile device. At the same time, with a nearby edge node, the device always sees the newest version of the app, but triggers an install on mobile device only when the app is actually being run, rather than when updates are being pushed out by app stores. Hence, end devices or users are not burdened by update handling.

AppSachets are an efficient app cache implemented as edge function to further reduce traffic in last mile due to apps and their updates. It comprises of two novel caching policies targetted at App ecosystem: (i) p-LRU which takes into account local app popularity and (ii) c-LRU which takes into account the cost of caching apps with edge function.

State simply, AppSachets provides answers to the following questions.

- How to best design an edge function whcih can cache apps and/or updates?

- How to efficiently use the limited resources (i.e., storage capacity) available to edge function to maximize hit ratio or minimize caching cost for Internet traffic due to apps and their updates?

- How to articulate cost vs. benefit of the edge function deployment on edge nodes?

To put this in context, When end user requests an app from an app store, the path app traverses include the following elements of Internet fabric to reach end client device:

1. App store or back end data center which houses apps uploaded by app developers respond either by sending the app package directly or a more commonly redirect clients to their content delivery networks. Similarly, the update notifications are delivered by app stores to devices via push notifications that trigger end user devices to request the app updates which get delivered in same way as app installs. This can happen automatically or involve users‘ approval based on end user preferences.

2. When clients request apps or their updates, the app packages are delivered from CDN cache which is either privately owned (e.g., Google CDN) or standalone (e.g., Akamai in case of iOS apps) from their geographically distributed points of presence.

3. The app packages from CDNs are transmitted over an internet typically operated by internet service provider (ISP) or Telecommunication companies via their network pipes. It is important to note that each request for any app or its update results in separate traffic flow in ISP pipes. This is first part of what is often referred as ‘the last mile’.

4. The second part of the last mile is the wireless connection (Wi-Fi or 4G) to which device is connected. The app or its update travels over wireless (Wi-Fi or 4G) connection to reach end client devices where it is stored and consumed by the end user.

Important to note here is that traditional CDNs cannot mitigate redundancy of traffic in the last mile simply because they operate on its edge facing ISPs as opposed to edge nodes which operate on the client side. In addition, the ISPs often opt to run their own CDNs to exploit redundancy in traffic passing through their pipes. Since, the apps and their updates are normally flagged as non-cachable content using HTTP headers, cookies etc., due to pay-per-download nature and issues related to intellectual property protection, these ISP CDNs are unable to mitigate the app traffic in the last mile. The pain point in the app delivery is the congestion in the last mile of internet. Edge functions operating past ISP near end users can potentially effectively leverage redundancy in traffic due to apps and their updates.

AppFlux and AppSachets are part of an end-to-end system that caters to end client devices and fits in the existing Android app ecosystem, without requiring any changes from app developers or any changes visible to end users. They complement the current app delivery model in ways that can remain invisible to end users and app developers.

2.2.2 Lowering app discovery barriers

While end users prefer native apps over web apps or other browser-based interactions [77, 32], due to their faster performance, seamless access to device features, security, etc., a wide gap continues to exists in app availability vs. their actual use. Simply put, with so many available apps, how can users search, install, and try them, even if they might benefit from or like them? This is called as app discovery problem. This is particularly the case for apps with a transient usage model, i.e., those used only in contexts users experience infrequently. For a developers thriving on innovation and quick rollout of new apps, this poses a grave concern, as it creates a high barrier for acceptance of their apps.

App discovery can be seen as two problems: (i) *contextual app selection* which refers to classifying and presenting relevant apps to users based on their current context, history of used apps etc., and (ii) *barriers to app's first use* which refer to effort on the end user's part to install and try out an app for the first time. In this research, we focus on the latter – reducing the barriers in app's first use by end users by leveraging edge functions. For contextual app selection, we rely on existing mechanisms employed by app stores, e.g., recommendation engines, user reviews, etc.

In response, we developed *Ephemeral Apps* as new types of apps that (i) 'pop up' instantaneously on end-user devices in appropriate contexts, (ii) are transient, in that they can completely disappear including the state they might have created from the device once the end-user leaves the context, (iii) do not require any additional effort from app developers, (iv) do not need to change user behaviour of how they use apps today and finally, (iv) offer similar levels of performance, access to device features and security as native apps.

Ephemeral apps make it easy for users to 'try apps out' before deciding to install them permanently, hence reducing barriers in app discovery; or they can use them only within their current contexts, then discard them, expecting to experience an updated app next time in the same context, prompting the use of new available apps by end users and aiding with app discovery. This eliminates not only the need to explicitly install apps that may only be of temporary interest to a user, but also 'de-clutters' end user devices from too many apps and avoids device pollution from their persistent on-device state.

Achieving the abovementioned goals, however, required both the use of *edge functions* as ephemeral app servers, and systems software support in Mobile OSes to support their execution. This build on top of AppFlux and AppSachets discussed above to extend support for (i) ab-initio app streaming support so that devices can directly run streamed apps, without requiring their installation and (ii) invisible app ephemerality to keep device clean of the apps and the associated state created on device without any app specific modification.

2.2.3 Maintaining relvance with newer user interaction models

Looking forward, end user interaction with web services is evolving. Once such evolution had led to the use of messaging apps: chatbots as web service consumption vehicles. Chatbots support very simple responses associated with end user context derived from the conversation. However, as of today they offers very limited functionality due to not being able to deliver executable content, and to then execute that content on the end user device. We posit that in order to realize the real potential of chatbots, apps or app like functionalities would have to be delivered inside chat interfaces.

There exist a number of technologies which can be used to make that possible. Examples include using an image with a link to a web app in the chatbot window, triggering an app that is already installed on the end user device, redirecting users to install an app, or showing a screen capture of an app running in a remote location [24]. These examples suffer from several limitations. All of these approaches require additional developer effort to re-factor existing services into finer granularity for delivery in chatbots, i.e., creating a functional partition of an online service or app that can be delivered directly via chatbot interfaces to end user. Further, there are trade-offs in using any of these approaches with respect to the cost of their data usage vs. the performance they provide in form of responsiveness and security/privacy concerns. These concerns were clearly evident in disagreements among app models to be used: web apps vs. native apps vs. thin client apps, and in costs associated with re-factoring of mobile web sites to mobile apps.

In response, we developed *AppInstant* edge function that automatically creates appropriate and lightweight *app slices* from native apps. With AppInstant, app slices can be used to deliver on-demand online services without requiring any additional developer effort. App slices run on end user devices and correspond to a single task that is carried out using an app. For instance, while booking a flight from a chat interface users would be presented with a screen that shows the chosen flight details and allows a booking action to be taken on it, in the same manner as if the airline app were installed on the end user's device. We prototyped AppInstant on Android. We use the term *task* to refer to a single focussed thing user can do with apps, and the term *activity* as a class derived from the Activity class of Android, which corresponds to the implementation of a single task.

AppInstant creates app slices from existing native Android apps using the standard Android SDK tools. A minimal app meta-data needed for presenting an app – an icon and a short text describing the app – are extracted from the app package and delivered to end user devices and rendered in the chat window. When a user launches an app slice, the app slice components are fetched via streaming. We call the support for streaming app slice as app slice streaming build on top of ephemeral apps. App slices can be executed on end user devices in the same fashion as a native apps. During execution, AppInstant tracks any on-device state created or modified by an app slice. When the user finishes, clean up is scheduled for the apps slices. We call this new tracking and clean up functionality as app ephemerality.

2.3 *Benefits and constraints*

Overall, we motivate the edge function based Android app delivery from its following benefits:

- **Efficient: Low pressure on backhaul.** Edge function based delivery of apps and their updates can leverage redundancy in their Internet traffic across different users making them an efficient choice.

- **Performant: Efficiency of native apps.** End user preference for native apps is, in part, due to their faster response times or better performance vs. that seen for web apps, thin apps and their consistent user interfaces mandated by the mobile OSes. Edge function based app delivery can keep these characteristics, by making it possible for native apps to be streamed from edge functions and run on devices with the efficiency of native apps.

- **Rich: Seamless access to device features.** End user preference for native apps is also driven by apps being able to seamlessly access device features for better user experience. Apps delivered from the edge function can preserve the benefits of native apps, and address web apps' limitations on access to device features, while maintaining the same levels of security as native apps.

- **Convenient: Interaction on-demand.** With so many available apps for multiple devices owned by an user, it is very inconvenient for a user to search, install, and/or update those apps. Edge function based delivery can reduce this inconvenience, by providing apps to users based on their connectivity to an edge function.

- **Minimal: Finer granular delivery of functionality.** To allow delivery of only the specific functionalities needed by end users for appropriate context while maintaining native app like performance, security and access to device features can allow app to be integrated with newer forms of end user interactions e.g., chatbots.

The resulting system must respect the following constraints to be a practical solution:

- **Invisible:** Delivery of updates to the end user devices must be invisible to user and driven by app's usage rather than their presence on end user device.

- **No change in user behaviour:** Using apps or slices delivered from edge functions must not present a learning curve for end users to use them i.e., users must be able to use them in same way as they use native apps installed on device.

- **Instantaneous:** Apps or slices delivered from edge function must be able to appear instantaneous in that a single ephemeral app should be able to appear on end-user devices within 300 ms. This rules out pushing full app installs on devices automatically and/or on-demand as the user goes from one context to other.

- **Transient:** Apps or slices delivered from edge function along with the state that they create on end user's device must completely disappear from the device once the end-user leaves the context. Important to clarify is that the intent is to keep device clear rather than removing the trace of its execution as was the goal of earlier work like [72].
- **No additional effort from app developers:** Delivering apps or slices from edge function must not require developers to change thier apps. The mechanisms and/or tools developers use to create must not be different from creating native apps. This is important from a practical standpoint of their acceptance by developer community.
- **Performance and device features:** Apps or slices delivered from edge function must be able to perform at same level as native apps and have access to device hardware on existing devices under existing permission models.

2.4 Measurements and studies

We start with presenting the measurements and studies carried out to quantify the pain points for Android app ecosystem and understanding Android apps to develop a practical solution.

2.4.1 Traffic due to app delivery

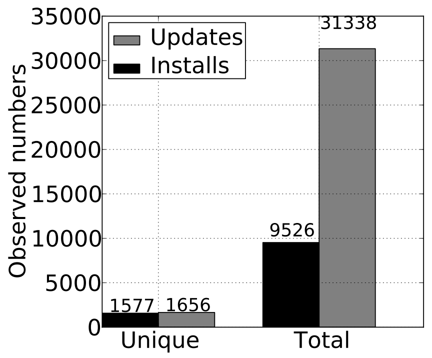
We collected data about all users in Georgia Institute of Technology, over an extensive, representative time period (from May 19, 2014 to Aug 21, 2014). The data is obtained from a network tap also used for security research, and is filtered to determine whether traffic was due to the installation or updates of apps from the Google Play store and includes apps and/or updates accessed by entire population including students, faculty and staff at our institution during the collection period.

Data Collection Methodology. App installs or updates are not directly identifiable in the traffic traces available to us. Instead, we observe that whenever a device initiates an install or update of an app from the Google Play store, this leads to a HTTP 301 response code from the store, which points to the location of the app within Google's CDN or server. This 301 response contains a location URL that points to the domain “play.google.com”, and contains the URL path element “/market/”. The URL also contains in its parameters the name and the version number of the app being installed/updated. The version information is either a single version number if installing a new app or if an app update results in removal of old version and installation of a newer one, or a colon separated list of two version numbers, i.e., the current and new versions. This information is sufficient for determining the overall set of IP addresses for which play.google.com resolves, as many portions of Google’s overall infrastructure (including app distribution) are served via their CDNs.

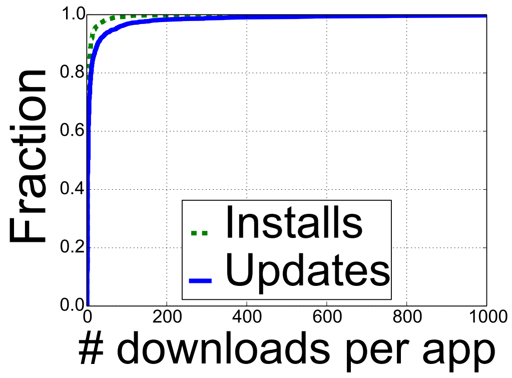
Prior to obtaining traces, we systematically resolved the IP for play.google.com over a multi-week period and recorded all resolved IP addresses. We configure our collection server with this IP information, to collect all packets that have any of these derived IPs in the source address section of the IP header and that utilize the TCP source port 80 (HTTP). After collecting all such traffic, we then use tshark to perform TCP packet reassembly, filtering out all traffic that does not fit the parameters of Google Play HTTP 301 responses. The resultant set of response codes represent all detectable Google Play app installs and updates for that two week period within our organization’s network. While traffic collection is ongoing, we use softflowd to generate netflow information for the network. At the conclusion of the data collection process, we use nfdump to read in, aggregate, and produce total bandwidth utilization for the time period of collection. The resulting measurements report a total of 2 Terabytes of 301 requests pcaps for this period from the Google Play Store. Unfortunately, updates and installs over encrypted connections (e.g., HTTPS) cannot be detected in this fashion and are not included in the data presented because the necessary information to detect an app install or update requires the contents of the HTTP 301 response code.

Observation in first glance

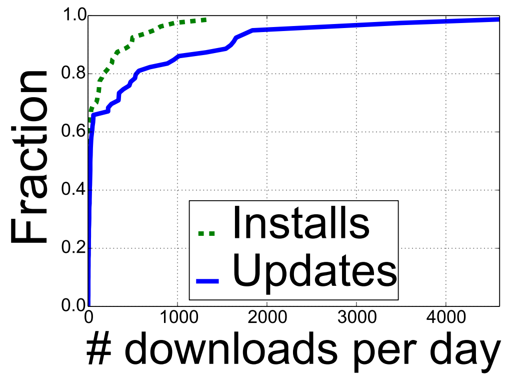
- *App updates dwarf the number of app installs.* Figure 2(a) shows that the total number of app updates is approximately three times the number of app installs. Figure 2(c) and Figure 3(a) show the distribution of app traffic aggregated on a per day basis. Figure 2(d) and Figure 3(b) show the distribution aggregated on a particular hour during the day. One might find it surprising that the number of installs also increases on the same days when updates are transmitted. This is due to the way we distinguish between updates and installs in the traffic traces added to actual installs. Also, we note that in some instances, updates are also



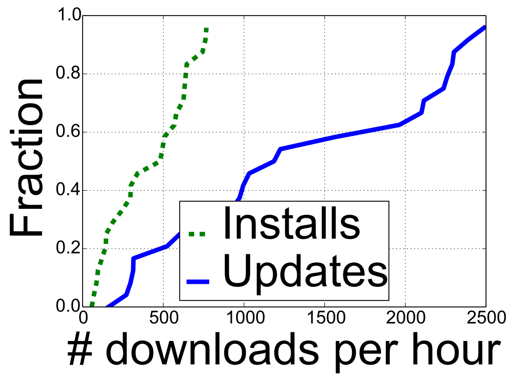
(a)



(b)



(c)

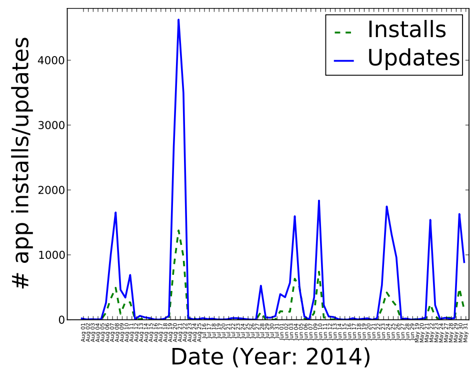


(d)

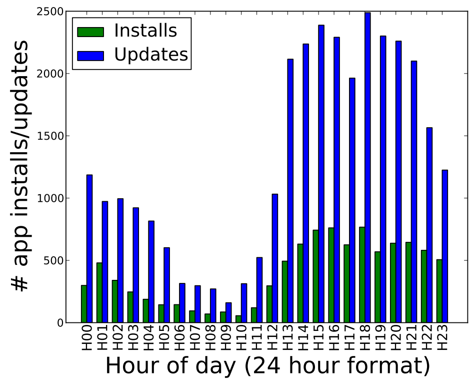
Figure 2: Android app traffic measurements over the period from May 19, 2014 to Aug 21, 2014 (excluding two weeks in July - a total of 11 weeks): (a) showing the total number of observed and unique app installs and updates; (b) CDF showing distribution of app installs and updates observed per unique app; (c) CDF showing distribution of app updates and installs for each day; (d) CDF showing distribution of app installs and updates at a particular hour during a day.

transmitted as full apps, if the diff between the updated and installed app versions is large. In any case, it is clear that app updates introduce significant traffic flowing in the last mile for apps resident (previously installed) on devices, regardless of whether those apps are actually being used or not. The long tail in the number of unique app updates and installs seen in Figure 2(b) (clipped on the x axis to highlight the lines) suggests a small edge-resident app cache would suffice to provide significant savings in last mile traffic.

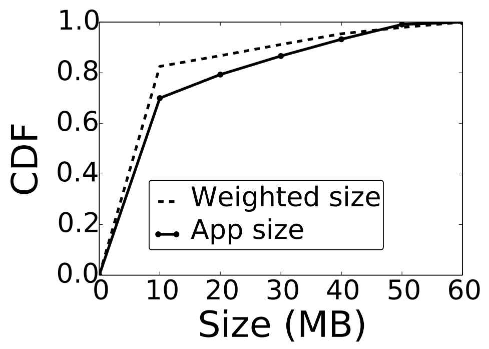
- *Bursty and cyclic nature of app updates.* Looking at the temporal patterns in the collected app traffic data, Figure 3(a) shows the app traffic observed per day. App updates show a bursty nature with respect to days with burst period of 10 days. This suggests in a significant period for edge functions to absorb updates. Variations in the absolute heights of the peaks can be



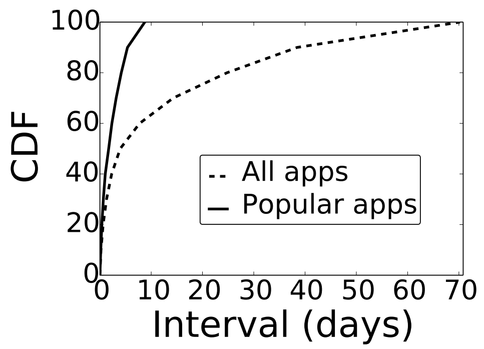
(a)



(b)



(c)



(d)

Figure 3: Android app traffic measurements over the period from May 19, 2014 to Aug 21, 2014 (excluding two weeks in July - a total of 11 weeks): (a) showing number of app installs and updates observed per day; (b) showing number of app installs and updates observed during hours of a day; (c) CDF of the size of the apps at the time data was collected; (b) CDF caching benefit i.e., number of days between successive app installs and updates observed in the measurements.

attributed to the variation in the number of devices present in our organization (from May 19th through August 15th) and the arrival of students in the fall (from August 15th). App traffic in a particular hour during a 24 hour period is shown in Figure 3(b). The diurnal traffic pattern suggests that even if updated apps have already been pushed to the app store by developers, actual updates coincide and are centered around the Internet rush hour (7-9PM) [27]. An updated app resident with an edge function can alleviate this pressure at peak times. The CDF of the number of app updates and installs per day in Figure 2(b) highlights the long tail distribution of app traffic with respect to days, i.e., there are a few days with heavy updates, but the majority of days are without any updates. Also, shown in Figure 2(c) is the CDF of app traffic per hour of day, to provide a complete picture of the data collected as part of this

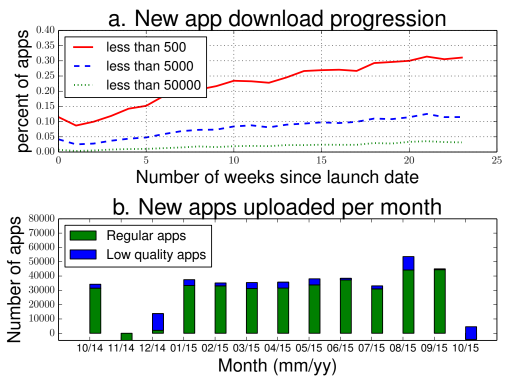
Table 1: Summary of measured traffic due to app intalls and updates.

	per app		per day		per hour		Total
	max	90%	max	90 %	max	90 %	
Installs	755	10	1377	420	217	11	9536
Updates (raw)	1288	19	4626	1540	443	44	31338
Updates (versioned)	895	20	1377	420	443	44	31338

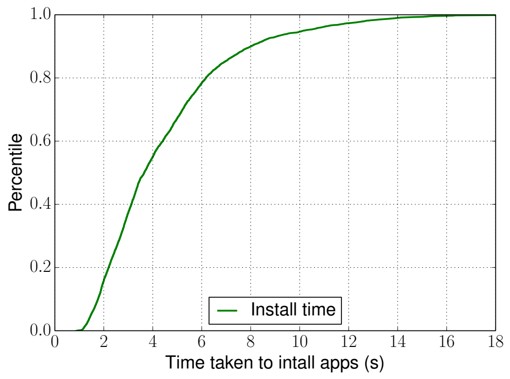
work. An interesting element is that app updates are distributed throughout the hours of a day which seems contradictory to results in Figure 3(b). But this is a result of the cumulative effect caused by combining hourly distributions during peak and non-peak days. AppFlux can alleviate pressure on peak hours and at the same time, reduce the interruptions seen by end user on their devices due to absorption of updates by the edge function which will be seen only if the app is actually used.

To assess the impact of app-related traffic on the Internet and assess the improvement opportunities that can be provided with better app caching poliecies, we further analysed the collected data collected discussed below.

A second glance at traffic. Table 1 summarizes our app traffic measurements. The results are further divided to show the number on app installs or updates observed per app, number of installs and updates observed per day, and finally, number of apps and updates seen during a particular hour of a day. Figure 4(a) shows the distribution of app sizes and the distribution of weighted app sizes where weights for an app is derived from its access frequency seen in our measurements. Figure 4(b) shows the distribution of app access with respect to interval at which apps and/or their updates are accessed shown for all apps and popular apps separately. We derive an app's popularity by ranking apps on their access frequency. It is clear that all popular app updates are finished within 10 days of the first roll out suggesting that the app updates occur in cycle of 10 days. We were not able to find an exact reason for this cycle, but intuitively, it is likely either due to app store's scheduling of app updates or a period arising out of different developers pushing out updates for their apps. In any case, this suggests there is a significant period for edge functions to absorb updates. Further, the difference in max and 90th%ile shown in Table 1, clearly highlights the bursty nature of app traffic in which app updates outnumber app installs by a factor of 3. The above observations bolsters our hypothesis about the benefits of caching app traffic on the edge. However, leveraging this redundancy in app



(a)



(b)

Figure 4: (a) App downloads in weeks after launch with more than 500, 5k, and 50k downloads (b) No. of apps uploaded to the Google Play store in 2014; (b) Showing CDF of the lower bounds on time spent during app installation.

traffic, to reduce congestion in the last mile, requires careful design of the app cache and caching algorithms. Regarding which we derive the following two hypothesis:

1. The gap between popular apps and all apps seen in Figure 4(b) leads us to hypothesize that a caching scheme that explicitly considers app popularity in its operation can provide the best cache hit ratio.

2. The difference in weighted app size and app size seen in Figure 4(a) leads us to hypothesize that a caching scheme based on (a) cost derived from storage and time an app resides with an edge function and (b) benefit derived from reduction in bytes transferred, can provide good hit ratio while limiting caching costs and hence, pave way for a cost model for edge cloud services.

2.4.2 Quantifying barriers to app discovery

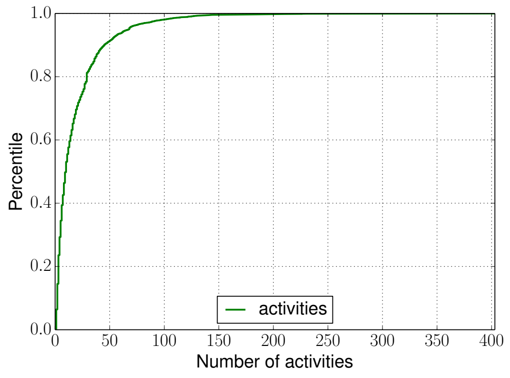
Concretely, a wide gap exists in app availability and their acceptance. In fact, only 15% of total available native apps are downloaded more than 5,000 times, as shown in Figure 4(a).(a)[3]. The app discovery gap is likely to widen with the impending explosion of new apps prompted by the wearables, on-demand interactions in Internet-of-Things, situation-specific apps, etc.

Further, we argue that automatically installing and uninstalling apps on mobile phones without

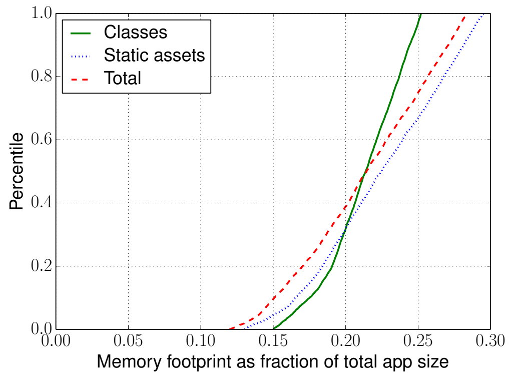
[3]Data source: http://www.appbrain.com/stats/

Table 2: Survey of popular chatbot SDKs checking their support to deliver executables and other content type.

Chatbot SDK	Executable content	Other content
Facebook	Javascript	text, images, web links
Pandorabot	None	text, files
Libre	None	text, images, web links, video, audio, animation
Telegram	None	text, images, web links, video, audio
WeChat	None	text, images, web links, video, audio



(a)



(b)

Figure 5: For top 5000 apps downloaded from Google play (a) CDF of number of activities in a single app; (b)CDF of memory footprint of the main activity in terms of fraction of classes, static assets and total footprint loaded when compared to total classes size, static assets size and total app size respectively

users' involvement can raise concerns about the permissions. Moreover, it takes a long time to install full apps as shown in Figure 4(b) which shows the time taken to simply install an app on an Android phone (Nexus 5) from a PC connected via USB cable using Android debug bridge (ADB), measured for 5000 apps - the top 100 in each category dowloaded from the Google Play store. Our posit that with ephemeral apps, this delay can be reduced by a factor of 10x, bringing the time required to try an app below 5 seconds.

We make another important observation that most apps turn on the network to access Internet anyways: out of top 5000 popular apps in the Google Play Store, we inspected, 98.9% of apps require Internet permissions. This highlights that using networked nearby resources for ephemeral apps puts minimal overhead on mobile devices. So in essense, ephemeral apps can be thought of as a edge function enabled device experience.

2.4.3 Supporting new user interaction models using apps: chatbots

The motivation of this work stems from the goal of minimizing cost, time and risk of creating a new developer ecosystem for chat bot based apps. We observe that messaging apps, drivers of chatbot based apps, are mostly used from mobile devices and users prefer native apps on them. Keeping this in mind, we asked the question that how and if we can deliver parts or slices of native apps to end user devices while maintaining the same experience as using native apps, without any involvement of app developers. We make the following observations:

- Apps contain more than one functionality. Figure 5(a) shows the CDF of number of activities per app over a period of for the top 5000 apps downloaded from Google Play store. Concretely, it shows an average 19 activities per app with minimum 1 and maximum 403 activities in an app. That doesn't include many ActivityFragments (a feature introduced in Android 3.0) in a single activity that app might be using to let end users perform many tasks from a single activity.
- Apps do not require to load all content available in an app package to offer a single functionality. Figure 5(b) shows the memory foot print of apps when launching the main activity vs. their total app package size which we observed to be always less than 25%.

2.4.4 Understanding Android apps

We decided to use Android to prototype app delivery from edge functions due its open nature providing us the ability to prototype a real system. We start with a glance on the lifecycle of an Android native app.

Life cycle of a mobile app.

To understand constraints in realizing an alternate path for app delivery, we studied the app's life cycle in the current app ecosystem, which can be summarised as follows. Developers upload their apps to app stores, which audit and check for security before making them available to authenticated end users. During installation, end users trust the integrity of apps from app stores based on self signed certificates by developer and agree to specific permissions required by the apps via a privileged app (Google Play or iTunes) deployed on end user devices. After installation, the app resides on the device's local storage, and is then available for launch. A launch involves loading its components from local storage, along with the mobile OS's app framework components, which are then collectively

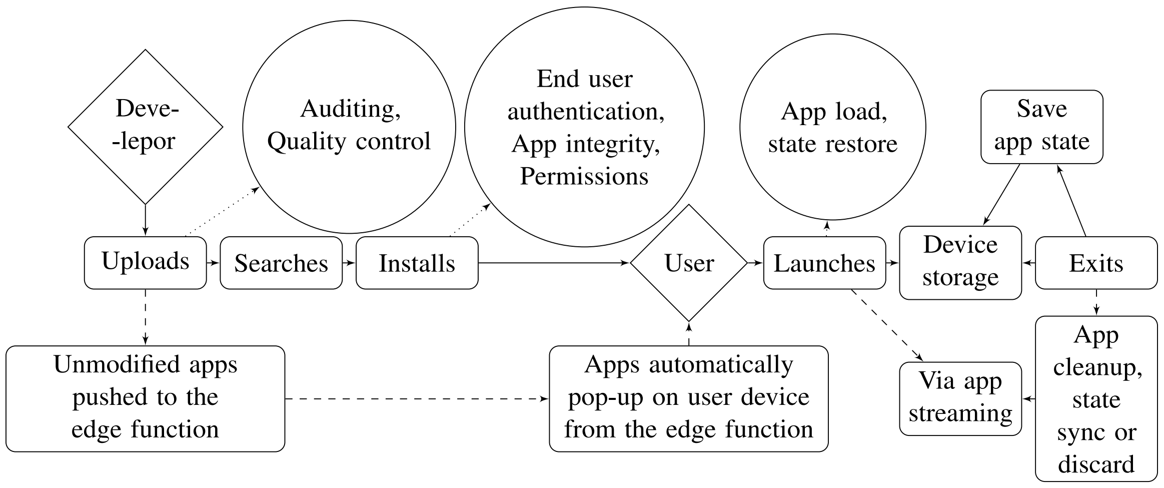
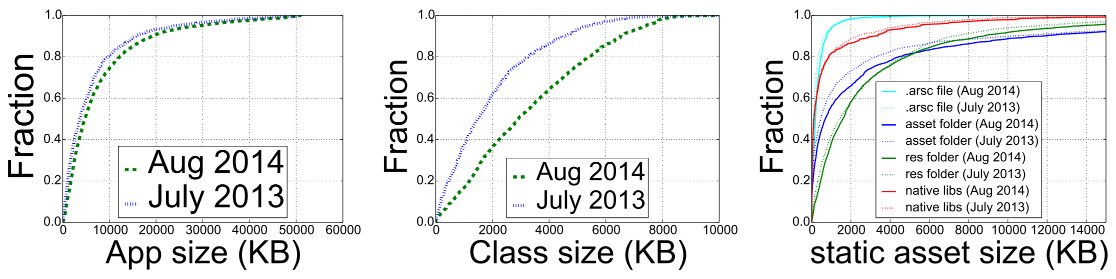

Deve-
-lepor
Auditing, Quality control
End user authentication, App integrity, Permissions
App load, state restore
Save app state
Uploads
Searches
Installs
User
Launches
Device storage
Exits
Unmodified apps pushed to the edge function
Apps automatically pop-up on user device from the edge function
Via app streaming
App cleanup, state sync or discard

Figure 6: The life cycle of an app in the current app ecosystem, with associated actions shown as solid lines. Dashed lines show the modification proposed for ephemeral apps. Dotted lines represent the system goals ensured by those actions.

run. Subsequent app runs leverage the app state created by previous runs, and resident on devices' local storage, to provide an efficient and personalized 'app experience'.

Preserving this experience, without asking developers to change existing apps, gives rise to a number of system-level differences compared to how apps are handled in the current app ecosystem. First, for apps uploaded to app stores, beyond carrying out actions like auditing and checking, the store proactively need to push apps to relevant, pre-registered edge functions. When a user devices connect with an edge function, the identity of the edge function need to be verified by end client device as a valid provider apps. After verification, for contextually appropriate apps, minimal app meta-data (as opposed to full app) required for presenting an app to a user should be sent to end user devices which then must be rendered in a fashion similar to that of installed apps, but on an 'different' home screen. To allow faster user access, app permissions must be granted lazily. When an user decide to launch an app made available by an edge function, app components streamed from it must be checked for their integrity individually (as opposed a single developer signature), and then, the app needs to be executed on end user device in similar ways as a native app. When the user disconnects from the edge function, cleanup is carried out for the apps not in the foreground, for their meta-data and any on-device state created by that app during its use.

Having established the high level vision of app delivery using edge functions, we now outline the technical challenges in form of answers to questions the are needed to achieve that vision. Although



(a) (b) (c)

Figure 7: App anatomy analysis for the Top 100 apps in each category available in the Google Play Store in July 2013 and August 2014: (a) showing CDF of total app sizes (apk file size); (b) showing CDF of uncompressed class sizes in apps; (c) showing CDF of other static assets (asset folder, res folder, arsc file, native libraries).

the discussion in this section is very Android-centric, because we prototype app delivery for Android app ecosystem but the concerns and design implications can be applied to other mobile OSes as well by carefully considering implementation details.

Diversity in app anatomy

Question: What is an Android app made of?

Answer: A typical Android app consists of app's configruation (Android Manifest.xml), executables (classes.dex, native libraries) and static resources (.arsc, UI images etc.). We observed two important aspect specific to mobile apps during our analysis. First, apps are self contained entities, i.e., apps rely solely on the Android platform features for their implementation. So, for an Android app, if functionality other than what is available on an Android platform is needed by an app (e.g., API, libraries, etc.), it is packaged in the app itself, and inter app communication is explicit (via API or Intent functionality), as opposed to it being a runtime dependency. Second, app components occur in two forms either complied together e.g., all classes in a dex file or .arsc file which can be decompiled using standard Android SDK tools to be recompiled to a finer granularity packages or are stand alone files e.g., UI images, XMLs, Android manifest etc. which can simply be included or left. Only exceptions are native libraries that have to be included as it is, if they are to be included in an app slice.

In order to determine the long term viability of app delivery using edge functions for Android devices as apps evolve, we undertook an extensive study by inspecting the top 100 Android apps in each category available in the Google Play Store, for a total of 4,969 apps. This inspection identified

Table 3: Showing app components of a typical android app.

App Component	Type	Role
AndroidManifest.xml	file	App configuration file used by Android
classes.dex	file	JAVA classes, libraries
.arsc		Compiled binary resource file
assets	directory	Static assets in a directory structure: images, fonts, textlables, etc.
res	directory	Locale, device, OS specific resources.
libs	directory	JNI libraries, any other native libraries used in apps.
misc	directory	Certs, HTML help files, app specific binary files, etc.

app constituents, but also aims to quantify their evolution over a year and finally, the impact of this evolution on the app update process and on app performance.

This is important because previous efforts geared towards application streaming for end consumers have been thwarted limiting their use by enterprise customers only, e.g., Windows application virtualization [30, 35]. This was attributed to the inability of the end user applications to achieve 100% compliance due to their run-time dependence on non-Windows (or third party) services. We found that by design, Android apps are standalone entities, running as separate (uid, gid) pairs, and rely solely on Android platform features for their implementation. So, for Android, if functionality other than what is available on an Android platform is needed by an app (e.g., specific API libraries, etc.), it is packaged in the app itself, and inter app communication is explicit (via API or supported Intent functionality). Therefore, run-time dependency is not an issue for Android apps.

Concretely, Table 3 shows the component in a typical Android app and Figure 7(a) shows the cumulative distribution function (CDF) of the total compressed size of downloaded apps, i.e., apks. It clearly show that app size is increasing over time, up to (but not exceeding) the 50MB limit imposed by Google. However, Google also provides two expansion files (2GB each) that can be used by developers to provide any additional resources required by an app. We conservatively analyze only apks and their components, despite the fact that any update on either expansion file also triggers an app update. Figure 7(b) and Figure 7(c) show the CDF of the apps uncompressed class size, and that of apps' other static assets. Figure 7(b) shows that app class size has increased for all the apps, hinting at increased app complexity, a trend expected to continue as more complex functionalities are introduced by newer apps. We limit the max value of the x axis in Figure 7(c) to clearly highlight that the sizes of other static app components do not vary much, with only slight increases. This does

Table 4: Android mechanisms for checking app integrity, maintaining sandbox and permission control.

Function	Mechanism
App Integrity	Developer certificates, Trusted source
Sandboxing	Discretionary access control (DAC), SEAndroid mandatory access Control (MAC)
Permissions	Check Binder IPC requests from apps to framework APIs based on UID, GID

not mean that these components do not change, but rather, those changes are not visible as changes in component sizes (e.g., changes in UI images or icons, etc.).

App integrity and permissions

Question: Which assumptions about app integrity and permissions are violated by delivering app slices on-demand?

Answer: Table 4 shows the various functions and mechanisms employed in Android. App integrity is established using developer certificates and check are implemented at binder IPC for granting/denying permissions to apps that enable them to use platform APIs to access device features at app installation time. However, these cannot be used for delivery of app slices as there is no installation step at all – i.e., to see an app, currently the full package of the app has to be delivered and installed on a user's device. In addition, the current mechanisms in Android for establishing app integrity at installation time use a single developer-signed certificate, hash of package contents and unique developer signature.

Native app execution and sandboxing

Question: What are the roles of different app components during app execution and how an app execution is sandboxed in Android?

Answer: In Android, an app executes as an instance of the Android Runtime (ART), identified via Linux's group and user identifier pair (gid, uid) allocated at the installation time. Sandboxing is enforced using the underlying Linux's discretionary access control (DAC) and mandatory access control (MAC) – SEAndroid policies associated with different type of apps. The device features are exposed via Mobile OS APIs, where permissions are enforced by checking the (uid, gid) pair of the requesting app and matching them with information saved at installation time. During execution, different app components are loaded from separate subsystems as described below:

- JAVA classes are loaded by the ART either as byte code from classes.dex in the app's package or as a mix of byte code and compiled binaries from the OAT file;

- Native libraries used in apps are loaded by ART using the Linux dynamic library loader;
- Static assets (e.g., form .arsc files or res or asset folders) and non-assets (e.g. AndroidManifest.xml) are accessed using the asset manager implementation in the androidfw app framework module;

To makes matters more complicated, these components can be used in different ways, e.g., an app can use either a URI and/or a binary buffer mode to load static assets. This also highlights an important point that existing mechanisms such as remote class loading or JAVA reflection do not suffice for delivering app slices on-demand.

Question: How to identify which app components contribute to a particular activity?

Answer: If we instrument all interfaces discussed above i.e., class and native library loading in ART, asset loading in asset manager and capturing the logs when a particular activity of an app is launched, we can identify which app components contribute to launch of a particular activity.

Native app state

Question: What persistent state is maintained for apps and/or created by apps?

Answer: Table 5 lists the state maintained for and/or created by an app during different phases of its lifetime. It is important to note that not all state is controlled directly by the apps. Instead, Android's system components are also responsible for maintaining app state.

2.5 Deduced design considerations

We discuss the design considerations for the design changes in Mobile OS central to use of edge function for app delivery:

- *Decoupling presentation from execution:* Before app streaming functionality can be designed, automatic identifiction and separating of the information needed to 'present' that app, app slice or update to end user device needs to be carried out. This is important to avoid too much overhead in just showing an app or an app slice to user device or informing device about an available update.
- *Designing app streaming:* For app streaming, all the app components (more than just executables) must be made available on-demand, via streaming, invisibly to the app being run. This

Table 5: Showing app state generated on device during app installation, execution and sync operations.

State Type	Phase	Location relative to /data	Access	Reason for persistence
On-device system	installation	app/*package*/base.apk	system	App package (.apk)
	installation	app/*package name*/lib	system	App native libs
		system/packages.list	system	UID, GID pair for apps
		system/packages.xml	system	Permissions
	execution	dalvik-cache/*architecture*/	system	Device-specific optimized app binaries
On-device App Private	execution	data/*package name*/	app indirect	To store app specific Shared document-like content [4], caches, cookies, data journals, database, shared preference etc.
	execution	data/*package name*/files	app direct	acessed via openFileOutput() or getFileDir() API
	execution	data/*package name*/	app direct	via getExternalFilesDir() API
	execution		app direct	store files on external storage via getExternalDirs() API
	execution	external sdcard mount point	app direct	to store files via getExternalStorageDirectory(), getExternalStoragePublicDirectory(), getExternalCacheDir() API
Remote	Sync	Cloud	app	explicitly committed to a cloud e.g., saved game [23]
Content	Sync	Cloud	app	fetched from back end, cloud storage or third party e.g., ads, etc.

needs to be carried out at levels of performance similar to those seen for installed apps which loade those components from devices' local storage. Furthermore, we need to augment the single developer based signature that is currently used to establish app integrity. This is not sufficient for streaming apps or streaming automatically created app slices which are loaded on-demand individually thus, making usage of a single developer signature futile. Further, lazy permission granting mechanism must be used to keep the time it takes for an end user to use an app to minimum. A similar feature was recently rolled out as runtime permission model for Android M. Finally, To create an app slice, once identified, app components needed to launch a particular activity in the app must be carved out and be made available on-demand in a chat window to provide a native app experience for that slice.

- *Ensuring correct and safe execution via streaming:* This poses the following technical challenge for executing streaming apps or app slices. To ensure that during execution, automatically created app slices are provided with the same execution environment as installed apps. This is hard because it violates a number of hard-wired design time dependencies of the Android runtime, app framework and the underlying Linux capabilities. For instance, when an app is launched, the name of the initial class which is needed by the runtime is extracted from an API exposed by the app framework. That information is stored by the package manager at app installation time exposed via an internal API. To ensure that streamed app or app slice execute correctly, the system state needed by an app needs to be populated at launch time. This includes assigning a folder to save app components or exposing a folder to allow for saving app specific state. Similar is true for permissions, Android sandbox and Android's internal framework components such as AssetManager.

- *Designing app ephemerality:* As a result of ensuring same execution environment, we also need a mechanism to clean up those system state changes and any additional state created by the apps and/or app slices. Currently, any such functionality does not exist in Android or other Mobile OSes. To support it, we need to track the changes made by an app or app slice or by system on their behalf. In a nutshell, all of the interfaces listed Table 5 need to be enhanced to track the state associated with an app slice. Important to clarify is that the goal is to keep device clear rather than removing the trace of its execution as was the described in earlier work like [72]

- *Automatically creating appropriate app slices:* To create an app slice, app components needed to launch a particular activity in the app need to be carved out from the app package. Then, they can be made available on-demand in a chat window to provide a native app experience for that slice to end users. The next important step is to create an optimal slice of the app components that fulfils requests for app components originating from different subsystems. This is hard because determining the optimal slicing is inherently dependent on how an app is used by end users. We need to profile apps by dry running them to identify app component needed for launch of an app slice or a single activity. Any further required app components as a result of app's usage are to be made available on-demand via streaming at runtime.

- *Preserving a native app experience for app slices:* Executing app slices as native apps gives rise to a number of system-level changes compared to how apps are delivered and run in mobile OSes. In the current install-and-run model, the app's presentation on a device (i.e., when its icon can appear on a device screen) and the app's execution are tightly coupled. When an app "appears" on a device, the full package of the app is delivered to the end user device, including code and other assets required for execution, it is verified using a single developer signature for the whole package, and the app is always executed in a sandbox once the requested permissions listed in the package are granted by end users during its first launch. But for app slices delivered in a chatbot, we need to decouple the presentation of an app from its delivery and execution.

2.6 Research Contributions

Concretely, the measurements and studies carried out as part of this work make the following technical contributions:

1. **Quantifying pain points of Android app delivery:** We presented concrete data to establish the pain points for Android app ecosystem that the solutions developed as part of thesis address. Further, this chapter establishes the important practical design considerations for edge functions those pain points for Android app eco-system.

2. **Unique app traffic characteristics demonstrating cacheability:** We measured Android app installs and updates at the network tap of our organization for 3 months period. To the best of our nowledgee this measurement is first of its kind and potentially the only one given that all the app traffic has moved to https since then.

3. **Tracking evolution of mobile app anatomy:** We presented data from a year long tracking of app anatomy for top 100 free apps in each category available from Android play store. This highlights an important point that the work presented in next chapter can be withstand the evolution of apps.

4. **Detailed description of app in Android ecosystem:** We described in details the life cycle of an android app along with system level mechanisms employed in Android OS to support them.

This led us to develop the practical solutions as presented in next chapter.

2.7 Chapter Summary

In this chapter, we presented measurements and studies about app delivery in Android app ecosystem. The rationale behind carrying out those was to find problems and opportunites which can server as concrete use cases to demontrate wide range of benefits of using edge computing based solutions. More generally, these studies and measurement highlight the importance of application specific logic which is crucial to definition of edge function. Furthermore, it highlights that newer system software support is required to harness the real potential of edge computing. Finally, these measurements and studies allowed us to quantitaively look at the problems and informed our the next step in our research to design, implement and evaluation of those solutions.

CHAPTER III

SYSTEMS SOFTWARE SUPPORT FOR APP DELIVERY FROM THE EDGE

Motivated by the observations presented in the previous chapter, we designed the edge function based solutions and developed new system software in Android to realize our vision of edge function based app delivery. In this chapter, we describe the details of their design and rigourous evaluations to quantify the benefits of using those solutions.

3.1 Introduction

We developed the following four egde function based systems developed to address paint points in Android app delivery.

- **AppFlux.** The volume of app installs and updates puts pressure on the existing app-delivery infrastructure due to interactions of end user devices with app stores via the Internet that involve app stores' data center, content delivery network's (CDNs) points of presence and the Internet Service Provider (ISP) pipes. 'AppFlux' is a novel app streaming approach to app delivery which reduces the load app delivery poses on the app infrastructure, and relieves users from having to deal with unnecessary updates potentially saving bytes on their costly data plan. By leveraging the emerging 'edge' tier of the Internet, the AppFlux edge function provides 'just-in-time' delivery of apps and their updates while (i) reducing the traffic due to app installs and updates seen in *the last mile of Internet* by up to 70%, (ii) facilitating twice as fast app delivery compared to CDNs and (iii) App streaming can potentially lead to 20% improvements in the app load times on devices. With its implementation for the Android app ecosystem, AppFlux achieves this in a completely invisible manner i.e., without requiring any changes by app developers or any explicit actions by end users.

- **AppSachets.** Data center based app stores offer users convenient app access, however, cause congestion in the last mile of the Internet, despite use of content delivery networks (CDNs) or ISP-based caching. With redesigned app caches – termed AppSachet – edge function based distributed caching can achieve a hit ratio of up to 83%, demonstrated on real-world Internet traffic due to apps. The redesign leverages proposed new caching policies, termed p-LRU and c-LRU, specifically targeted at limited storage available at edge nodes and for the traffic caused by app installs and updates. A cost benefit analysis shows that the additional cost required to deploy AppSachet as edge functions can be recovered within the first three months of operation.

- **Ephemeral Apps.** Despite a tremendous increase in the number of mobile apps, coupled with their popularity with consumers, there exists a wide gap in app availability vs. their use. Recent trends suggest that this gap will further widen in the future. *Ephemeral apps* delivered from the edge, lower the barrier for end-user app acceptance by removing the app installation step when 'trying out' new apps, without requiring modifications to current apps or any additional programming efforts by app developers. We estimate the resulting reduction in time-to-use for apps to be a factor of 10x by providing then from an edge function.

- **AppInstant.** The immense popularity of messaging app and advances in AI are posing chatbots as the main vehicle for online content consumption. For the chatbots ecosystem to succeed, a new developer community needs to be mobilized to develop apps and re-factor content, which is both costly and difficult to achieve. Further, chatbots SDKs will take more time to mature. Currently, only content can be delivered via chatbots, and not functionality which would be needed for the rich types of interactions users are accustomed to. In this paper, we present an alternative - AppInstant - an approach which allows the chatbot ecosystem to piggyback on the native app ecosystem, and deliver new functionality to end users while allowing app developers an opportunity to expose their apps more often. To ensure that AppInstant doesn't require any additional developer effort for either refactoring the content or for development of new apps, we propose the notion of app slice. Prototyped using Android, we show that it is feasible to automatically create app slices from existing Android apps and to use them in same manner as native apps installed on end user devices. Delivery and execution of app slices on end user devices, however, requires additional support for app streaming, runtime integrity checking and app ephemerality, which we implement for Android. We show experimentally that AppInstant's app slices perform better than other technologies such as web apps, thin clients or on-demand native app delivery. Further, we show that using the emerging 'edge tier' infrastructure, app slice can perform better or at same level as installed native apps.

In designing the above solutions, we made some high level and low level design choices. We discuss those choices next.

3.2 High level design choices

We start with discussing crucial high level design choices for our new vision of app delivery:

Edge function vs. cloud service. Mobile devices are subject to well-understood constraints, such as power consumption. Using remote resources not subject to those constraints can, enhance user experiences on devices. Concerning the use of such remote resources for ephemeral apps, we argue the relative advantage of edge functions vs. remote data center based services in terms of (i) increasingly fast wireless networks, (ii) cheap storage to house app repositories and low access latency due to lower network distance.

Type of apps. Currently, all mobile OSes like Android, iOS etc. support the following app models:

- Native app: specifically designed to run on a device's OS, uses features exposed by platform via APIs and is constrained by OS semantics.
- Web app: the entire app or its parts are downloaded from the web during execution. It can be accessed from any web-capable mobile device without requiring operating system-specific customization.
- Hybrid apps: most native apps utilize web connectivity, and web apps provide offline modes. The resulting form of apps are referred to as hybrid apps.
- Thin client apps: require a remote access app, e.g., VNC client on device while actual execution happens remotely.

We posit that streaming ephemeral apps can potentially obtain native app user experience in terms of responsiveness, access to device features, security, efficient operation than thin client apps with central management capabilities of web apps by allowing on-demand streaming of the native app components needed for execution. We choose to use native apps to prototype them for end user prefer them over other forms of apps e.g., web apps, thin apps etc.

System layer for app streaming. Our goal is that functionality required to support ephemeral apps must leverage the existing app model supported on client devices. Modern devices with 'tall' mobile OS stacks, and 'thinner', well packaged mobile apps on top. We posit that since the majority of system level functionality required by apps is abstracted by the app framework APIs, runtime, etc., provided by the mobile OS, it is possible to hide the complexity associated with app streaming and ephemerality completely from app developers, while providing efficient app execution leveraging devices' capabilities. On the edge function side, this also allows to leverage system level information like the order of app components to optimize streaming performance [57].

3.3 Low level design choices

Having made the high level design choices, there are a number of design decisions that we are faced with considered in designing the three main elements for Android i.e., Ab-initio app streaming, app ephemerality and creation app slices that made the new vision of app delivery possible. We discuss them below.

3.3.1 Ab-initio app streaming

Designing ab-intio app streaming requires designing two main mechanims to be designed. We discuss the design choices and decisions we made to design those below:

3.3.1.1 On-demand delivery

Design choices:

Network class loaders: For streaming apps or slices, app components are requested from different subsystems and accessed in different ways as discussed earlier. This rules out the use of network class loaders for delivering app slices on-demand simply because not everything is loaded as a JAVA class in an Android app.

Network file system: Another design choice would be to mount a network file system partition on a folder on the end user device which houses app packages. When an app slice is launched an app can load the required components from that folder and also during subsequent streaming app execution. It is technically feasible but it would require the app mounting the partition, in our case a messaging app, to be assigned root access on device which is simply not a good security decision. Further, mounting a network file system is possible without root privileges by using FUSE drivers to implement a new user level file system. But that means that the interactions with the file systems need to be implemented explicitly anyways. Further, it would require making sure the existing app security model and permission model isn't violated in using that partition which would lead to changing the app framework anyways. So, it is better to confine the changes to app framework. Also, since most app components are files mapped and loaded during execution, relying on network file system to optimize the app component transfers makes it impossible to exploit opportunities to reduce the device side overheads during app slice execution without changing how these components are used explicitly in many places in Android. For instance, it becomes impossible to defer the class/asset look up overhead to the server, or to use server side uncompression of app packages shown to provide better app loading performance [57] without explicitly changing how class look up is performed in ART or asset is read using zipped data stream derived from app package vs. uncompressed app components available via AppInstalnt server. Finally, it makes it difficult to track those fetched slice components as the caching semantics are controlled by the network file system.

Design decision: We decided to implement delivery as a two phase process spanning messaging app, Android's app framework and app runtime. In the *presentation phase*, when a chat bot decides to make a functionality visible to an end user, an manifest is delivered to the end user device. At this point, end user device can display app or app slice as an icon with a short description of what can this be used for to end users. Till this step, there is no involvement of Andorid's system components. When a user decides to launch an app or app slice, the same interface as native app for that online service is shown to the end users by fetching the app components listed in manifest from AppInstant server and launching the app or app slice as if it were an installed app on end user device. Then, additional app components may be required if a user tries to do anything with that app or app slice example, say launch another activity. So, those app components are made available on-demand via streaming from app server where they are present in an uncompressed copy of the native app package. For further execution, any other app components are made available on-demand and checked for their integrity at run time.

3.3.1.2 Streaming execution

Design choices:

New vs. existing API: The choice is to integrate handling of app slices in existing Android APIs vs. creating a new API to launch app slices. This is important because if we add support for app slice in existing APIs, it will result in undesirable additional implementation complexity and overheads in launching and execution of installed native apps for operations that are specific to app slices e.g., runtime integrity check, etc. Further, this is the integration point for chat bots as well, i.e., how will chat bots launch app slices on an end user devices. Keeping the same API can cause conflicts between installed apps and its slice delivered from chatbots.

Design decision: To keep things clean and simple for an app slice, messaging app uses a newly added system API - startActivitySlice()- via its Android context. Android content is an interface to global information about an app environment and can also be used to launch other app's activities via startIntent() API. The *execution phase* starts when a user launches an app slice by touching its icon. The AppInstant support in Android includes provisions to create and maintain a sandbox environment for the app slice, similar to that of an installed app. To ensure compatibility with existing apps, this involves assigning appropriate permissions, applying the app slice security context, and forking a new instance of the enhanced Android runtime. Specifically, before forking an instance of the Android runtime, the Android Activity Manager asks the Package Manager to assign to the instance ephemeral uid, gid identifiers, that belong to a separate range from existing uid, gid ranges used for installed apps. They are later used to handle permission exceptions to provide lazy permission approvals for app slices (e.g., if an app requires more permission to access a device resource). To keep app sandboxing guarantees for app slices, we added a new SEAndroid security context and associated policy: app slice domain. During execution, all additional requests to load classes, native libraries and all static assets originating in runtime's class loader, native library loader and the app framework's asset Manager respectively are intercepted and redirected to the AppInstant's app slice servers via a streaming client and runtime integrity checker discussed next.

For native apps, app components are stored on device's local storage at a location assigned at installation time which is not changed till the app resides on device. For app slices, there is no such location and it has to be created and managed at runtime. by intercepting all app component requests, streaming client stores the fetched app components on appropriate storage locations. These app components can further be cached and reused during execution of an app slice for better performance.

3.3.2 Invisible app ephemerality

Design choices:

Confine apps: It can be argued that confining accesses granted to running app or slices to specific directories should suffice for tracking state. But, not all app state is maintained at a file or directory granularity. For instance, permissions added for an app are appended to a file /system/packages.xml which if removed or corrupted would break the system.

Complete tracking and removal: Approaches such as those described in forensic deniability [72] can be used to run app slices in private sessions, monitor its IPC and track all its memory allocations as well as interaction with any device features to track and remove every possible trace of an app execution. However, it is important to note that Android app are already run in an sandbox and sandboxing a sandbox would entail too much performance overhead. It is important to remember that our goal is to keep a device clean as opposed to providing strong guarantees. So, we can minimize the overhead associated with tracking and avoid device pollution by using a more light weight solution that can rely on reporting by Mobile OS rather than invisible tracking.

Design decision: AppInstant keeps end user devices clear of the system and app slice's private state created for or by its delivery or execution. To facilitate this, hooks are added in relevant places in the app framework, runtime or libraries, as listed in Table 6, to explicitly log the on-device app slice state on a per app slice basis using additional library-supported APIs. These per-app slice logs, implemented as SQlite's databases, are key to enabling system level app ephemerality guarantees and are handled by an ephemerality manager, which is part of a new system library: libephemeralutils. The ephemerality manager, allows processing of the logs under different policies, each of which is derived from end user preferences about the state created by app slices. For instance, if a user indicates that he wants to discard all app slice state then, on app slice exit, the ephemerality manager accesses the logs to gather the app slice state generated by that slice, to delete such state, and to ultimately remove the logs from the database. It can also be argued that instead of a per app slice ephemeral log, a single log would suffice for all app slices. However, that prevents users from being able to specify different policies of how to handle removal of the state for different apps. For instance, for a shopping app's slice, end user could be allowed to choose that he wants to remove all the app components for that app slice but still keep his shopping cart or invoice created by that app slice on-device. Further, libephemeralutils also implements checks to ensure integrity of each received app slice component from the streaming client using the top level merkel tree hash shared during the presentation phase, thereby ensuring runtime integrity check.

3.3.3 App slicing

There is a number of possible ways to restructure apps or to create app slices from app packages.

Design choices:

On code boundary: One simplistic approach would be to partition the app based on the static code boundaries, i.e., classes in case of Android apps. However, launching apps not only requires classes but also other static assets, ruling out this approach.

On activity boundary: Another simple approach would be to create app slices as bundles of classes and assets that are part of any particular activity, for instance those referenced from the main activity class. However, as it turns out for most apps, the activities defined in typical Android apps are tightly linked to each other or have some form of reference to the main activity. Hence, for every activity the corresponding bundle would be almost equivalent to the entire app package, leaving out only classes and/or static resource which are unique to the device configurations rather than the app's functionality itself (e.g., leaving out the UI images for different form factor of devices).

On execution boundary: In the approach that we are using in AppInstant, the partitioning is carried out based on the classes and static assets that are loaded at run time when an activity is launched and before it starts servicing the user input. Detecting this condition is simple – launch an activity and do nothing. Except for the apps that keep doing some task in background or implement Android services, there are no requests for classes and/or static assets during this period. It is important to note that apps with background services are also not good candidates for being delivered as slices anyways because app slices are intended to augment the experience that chatbots provide based on users' explicit intent rather than a continuous task. For instance, there is no point of delivering a footstep counter app as a slice instead it should anyways be installed on the device.

Design decision: On the AppInstant app slice server side, we create an execution dependency set for each activity in an app. This is actually a subset of all the app components available in the app package using static and dynamic analysis of app packages. Specifically, we extract this subset by launching app's activities on an instrumented Android running on a virtual machine, enabling us to profile them and thus create a lists of app components associated with different activities. Once, this list is created we use the standard Android SDK tools to create multiple packages – or app slices – from the same app package. It is important to note that this has to be carried out separately for each device with a different form factor because for different form factors supported in a single app, even though the name of resources may be same but they may point to different directory in app package and are resolved at run time by asset manager. This is not a hard technical limitation but of our current implementation which can be handled simply by running the same process for different form factor devices and creating app slices per form factor. Additionally, we create an *app slice manifest* for each app slice which contains only the app components required to make the app slice appear on an end user device, so that it can be subsequently launched. The slice manifest is extracted from the native app package without any developer effort and contains its package name, label, main activity name, link for app's icon and the top level hash of the merkle tree of uncompressed app slice components as leaf nodes. The actual app components (classes, static resources, etc.) required to run an app's activity are only fetched at the launch time via streaming.

3.3.4 App caching policies: p-LRU and c-LRU

We developed two novel cache policies for managing the cache of apps and app updates using AppSachet edge function. Policies are specifically defined based on opportunities observed from the app-traffic characteristics captured in our measurements. The two policies – p-LRU and c-LRU – are described next, and the overall description of the cache management operations with either policy follows the same operating flow illustrated in Figure 8.

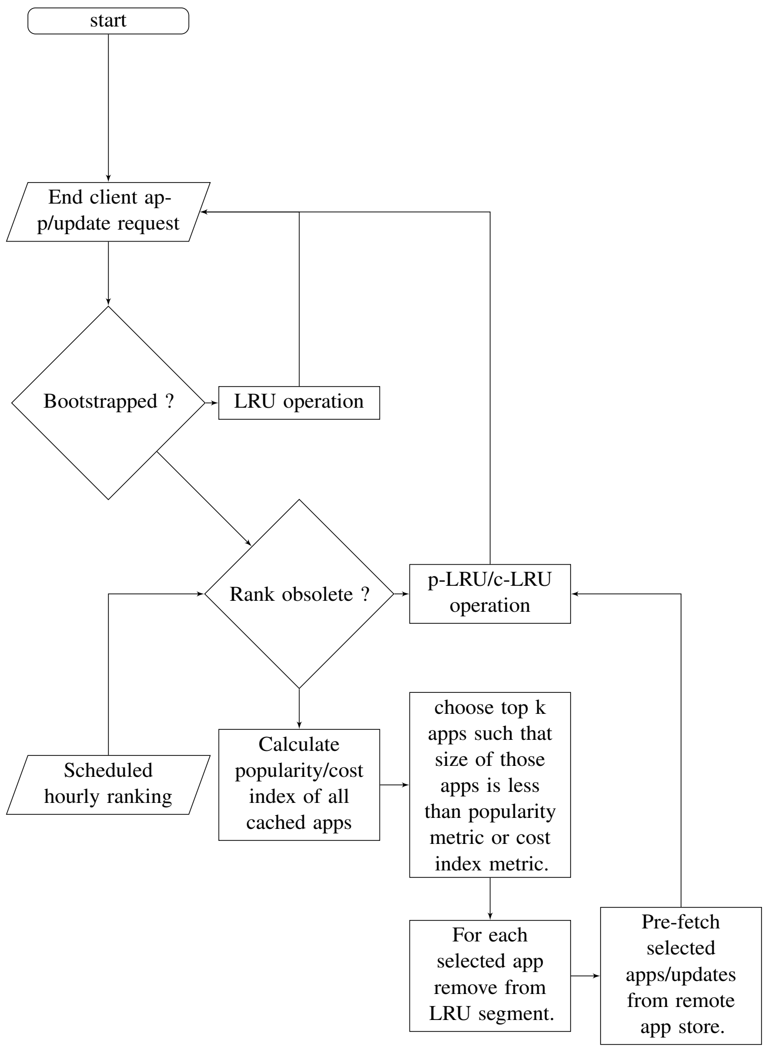

start
End client ap-p/update request
Bootstrapped ?
LRU operation
Rank obsolete ?
p-LRU/c-LRU operation
Scheduled hourly ranking
Calculate popularity/cost index of all cached apps
choose top k apps such that size of those apps is less than popularity metric or cost index metric.
For each selected app remove from LRU segment.
Pre-fetch selected apps/updates from remote app store.

Figure 8: Showing the operation of AppSachet edge function.

3.3.4.1 Popularity-aware Caching: p-LRU

p-LRU cache is designed to operate based on app popularity, observed as an important characteristics of app traffic. p-LRU divides the available storage space for caching in two parts: (i) LRU based and (ii) popularity based. The size of each segment is decided based on popularity metric which is defined as percent of storage space allocated to popular apps on an edge node. p-LRU cache is similar to a segmented LRU (SLRU) [102] in the way it keeps two separate segments of cache, but differs in the eviction strategies in the LRU-vs. the popularity based segment. The p-LRU cache works as follows:

During p-LRU bootstapp period, e.g., the first 24 hours, p-LRU acts as simple LRU. After that, apps are ranked according to the apps that were accessed in the past 24 hours based on the number of times they were accessed or *popularity metric*. For instance, if we have a cache of size 1 GB, and we see that 40% of apps are being accessed repeatedly, we set popularity metric at 40%. This will result in reserving 40% storage space, i.e., 400 MB, for storing popular apps and 60%, i.e., 600 MB, for storing recently used apps.

The popular apps and their updates are then pre-fetched until the popular segment is full. If the app is present in both LRU and popular segment, it is kept in the popular segment, so that LRU can accommodate more apps. Note that there are many apps that although not popular, not caching them would result in a considerable reduction in hit ratio, also highlighted by the gap in all apps and popular apps in Figure 17(b). Since, there are considerable number of apps that are often not popular but not caching them would result in a considerable reduction in hit ratio also highlighted by the gap in all apps and popular apps in Figure 17(b). Once p-LRU is bootstrapped, app ranking is repeated every hour and popular apps are pre-fetched for that hour.

3.3.4.2 Cost-aware Caching: c-LRU

Similar to p-LRU cache, **c-LRU cache** divides the available storage space for caching in two parts: (i) LRU based and (ii) cost of caching based. It uses a cost index to quantify the cost of caching an app on edge node. Intuitively, the cost of caching can be derived from the following metrics: (i) The number of times it is downloaded when compared to all the apps downloaded from that edge function or the *download ratio*; (ii) the time for which a particular app is kept with edge function's storage compared to its first download or *utilization ratio*; (iii) the time an app has already spent in the cache without actually being requested by end users or *recency ratio*; and (iv) the size of the app that needs to be stored. e.g., if any particular app whose size is 50 MB and is accessed 10 times and we have two other apps whose sizes are 20MB and 30MB, and are accessed 5 and 8 times respectively in the same interval, then we should give preference to caching the two smaller apps than one large app. One exception to this rule is that c-LRU must handle updates and installs separately because updates are always smaller than installs and this would lead to installs never being cached on edge node. We started with giving equal weights to each metric, and the value of each is normalized i.e., varies from 0 to 1. The app with the lowest cost caclulated this way is considered the most suitable one for caching at an edge node. After experimenting with different combinations of weights and metrics, we zeroed to the below mentioned definition of cost index of an app stored as:

Cost index = $[DR * (1/Appsize) + UR + RR)]^{-1}$

where,

Download ratio (DR) = number of downloads of that app / total number of downloads

Utilization ratio (UR) = hours spent in cache / hours since first download

Recency ratio (RR) = 1/hours since last download

The lower cost index results in lower cost associated with storing and hence, higher benefit, because the app may be accessed too frequently or uses very little space or a combination of both. The c-LRU cache works as follows: during c-LRU bootstrapp process, i.e., the first 24 hours, c-LRU acts as a simple LRU. After that, apps are ranked according to the cost index of apps accessed in the past 24 hours. The segmentation, pre-fetching and eviction in c-LRU work similarly to p-LRU except the use of cost index vs. popularity.

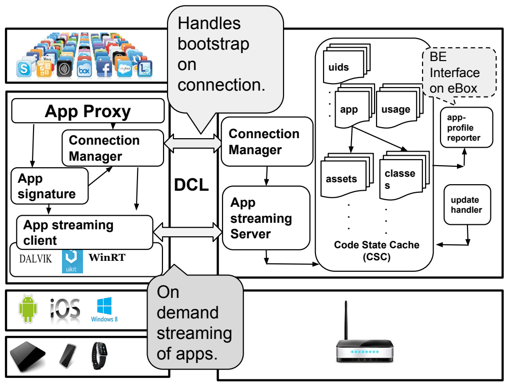
The cost function described above tries to maximize the utilization of resources used by edge function. However, the model permits for additional cost functions, including ones that incorporate consideration of different value generated from different apps. The ability to attach a value to an app in case of AppSachet or generally an edge function can pave the way to creating a quantifiable economic model for the upcoming ’edge cloud’ infrastructure, a concern of utmost importance regarding edge cloud deployment, which hasn’t been addressed in any of the recent edge cloud research [70, 100, 106, 60].

3.4 Design and implementation

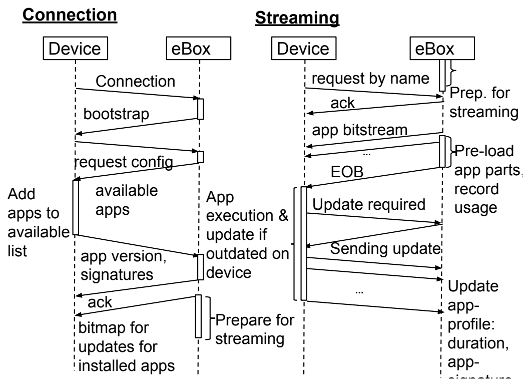
In this section, we describe the design and implementation details of the four concrete systems developed as part of this thesis.

3.4.1 AppFlux

AppFlux encompasses the edge function-remote app store interfaces needed to obtain updates. The outcomes of edge function interactions with a remote cloud or app-store is that edge function locally cache those apps that are more likely to be used by nearby mobile devices. AppFlux complements



(a)



(b)

Figure 9: AppFlux Design: (a) AppFlux device-edge function interface; (b) DCL protocols for connection and app streaming.

existing mechanisms for "on device" app installs and/or updates with instant edge function-based, just-in-time streaming of app executable, whenever apps are run. A desirable side-effect of running an app from the nearby edge node is a consequent update to the app resident on the device, but as this is done asynchronously with the app's execution, end users do not directly perceive such an update, nor do they experience the delays of update actions (they will see the battery power consumed by such updates, however). AppFlux operates by relying on device-resident functionality to transparently intercept requests for app components like classes, static assets like images, XML, etc., and then redirect such requests to network-accessible edge functions or to the device's local storage. The latter is so that other devices operate just like today's devices, relying on updated apps stored locally. Devices enhanced with app streaming, however, can use apps stored in the accessible edge function, with the desirable outcome being that users will use the newest app version currently available with EF. In case of disconnection while streaming an app, there are two cases that arise: (i) subsequent usage of the app requires only the already fetched app components in which case user would not see any interruption and (ii) subsequent usage requests an app component which hasn't been streamed yet. In that case, user will see an interruption and would have to restart the app.

The realization of AppFlux developed in our work addresses the technical challenges enumerated earlier, with a design that splits the app eco-system into two components: (1) a front end end user device – edge function interface, and (2) a back end component interfacing the edge function with a remote cloud or app store. As shown in Figure 9(a), the main elements of AppFlux are (i) the

AppFlux Client – embedded in app framework of mobile OSes, (ii) the *AppFlux Server* – providing on-demand streaming of app components, and (iii) a *Device Cell Link (DCL)* – that comprises a protocol for bootstrapping and app usage reporting which then, drives the prognostic adaptation in app streaming shown in Figure 9(b). For completeness, (iv) we also sketch the expected interactions between edge function and the remote cloud, such as populating app caches (push- or pull-based) and app-profile consolidation, but defer the discussion on caching policies, governing policies about how app executable, static assets, and associated app state are managed – for next section of this chapter.

The implementation of AppFlux uses either available Android platform components or open source technologies. Specifically, (i) the device-side AppFlux client elements are implemented as a patch for Android (specifically, modification and/or enhancements in cutils, dalvik, androidfw, and init.rc), and (ii) the edge function-resident elements are implemented using the node.js TCP socket API. (iii) Additional elements of the edge function-app store interface can be implemented using HTTP API exposed by app stores like Google Play.

3.4.1.1 AppFlux Client

The AppFlux front-end component provides the device-edge function interface and is designed as a split module, one half residing deep inside the mobile OS and the other resident on an edge node, as shown in Figure 9(a).

***Operation Duality* enabling Connection Manager.** The connection manager is responsible for communication setup when an end device first connects to a app streaming edge function. This entails (i) exchange of a list of installed apps, their versions and signatures, and (ii) exchanging the streaming server's configuration, i.e., port number, IP address, and the available apps. In the current prototype, a file in the Android file system, i.e., /data/AppFlux/app-list, is used to store a list of apps available on the currently connected edge function. On-device population of this information is implemented as a patch in Android's cutils exposing a wpa-supplicant listener that raises a new Wi-fi connection notification which then triggers AppFlux's bootstrap process.

***Prognosis Enabling* App Signature.** App streaming maintains a concise representation of each app, termed *app signature*. It is a list of the app's executable components and/or assets that are loaded during its execution on the end client device. The device-resident app signature module is responsible for maintaining these signatures for all device-accessible apps. In Android, this is achieved by keeping a list of classes and/or static assets to be loaded during app execution. The signature is stored as an app-specific file in the Android file system, specifically, in the configuration folder for app streaming on the device, i.e., /data/AppFlux/*app-name*/signature. App signatures are shared with edge function via the connection manager when the device first connects to an app streaming edge function. This is used by edge function to adapt the streaming content to include components that would be needed by that app by a particular user.

***Operation Duality and Sync Agnosticism* in App Proxy.** In a streaming-enabled device, the app proxy is responsible for intercepting executable and/or static asset requests (at the app run-time layer) and redirecting them to the network or to local storage, the latter based on connectivity information in the connection manager that actively monitors the device's connectivity to an edge function. Such interception makes it possible for app streaming to be transparent to app developers, i.e., no changes are required to existing apps. The app proxy also interacts with the app-signature module, to store and/or retrieve history-based app signatures. In our Android prototype, the app proxy is implemented as a module in the Dalvik VM (for app classes) and the androidfw module (for static assets). In Android, when an app requests a class via JNI, it is loaded from the classes.dex file or from dex-optimized classes cached in an app specific directory. When a static asset is requested, it can be loaded from the multiple locations, i.e., res folder, arsc file, asset folder, icon directory, etc., present in the app's apk installed on the device after passing through an unzip data stream. A URI and/or buffer mode can be used for loading static assets. The app-proxy spans Dalvik and the Androidfw library, and forwards request to the app streaming client or uses existing loading mechanisms to load from local storage. App-proxy provides URIs for classes, e.g., com.adobe.reader. *classname*, which are identical to those specified to the class loader by its JAVA framework or app's locale, vendor information and asset's name, to allow edge function to correctly select and stream required assets, as explained in the next sections. With this design and implementation, since there are no changes in the interfaces used by apps, even for static asset loading, app streaming operates without requiring app changes and invisible to end clients. The app proxy is also responsible for handling update triggers from edge function. On the trigger, it uses existing interfaces (i.e., Android's package manager) to trigger app update on device. The incremental updates when delivered from app stores are consolidated at edge function for streaming but are delivered as incremental updates only to devices which are handled in the same way as they are handled by existing devices.

***Invisible* App streaming Client.** The app streaming client is simply a TCP socket client that accepts requests from the app proxy and then handles app streaming responses from the server. It is embedded in dalvik and the androidfw library to cover all app components. TCP sockets are used (vs. HTTP) to minimize connection and packet overheads during streaming. The implementation modifies Android's cutils to expose APIs to dalvik and androidfw library that allow it to request classes or static assets.

3.4.1.2 AppFlux Server

***Sync Agnosticism* enabling App streaming Server.** The app streaming server is an edge function, listening to requests from end client devices for app executable and static assets, on a port number exchanged during connection setup. It is implemented as a node.js-based TCP socket server, and interfaces with an in-memory cache for classes and static assets – part of the code-state cache (CSC). It translates class names to directory entries, which in turn are pre-populated based on edge function-app store interactions, where apps are de-compiled on the edge functon into the same directory structure as the source. The loading of static assets is more complex, because (i) there are more locations to search for an asset including the asset folder, the .arsc file, the icon directory, xml files, etc., (ii) the same asset may refer to a different file depending on locale information, and (iii) some apps may choose to load assets in a raw buffer and parse individual assets for performance reasons. This leads to some additional computational overhead imposed on the app streaming edge function – rather than the end client device – to maintain an index and to search and load static assets. An important server action is to compare the app version on the device to the one resident in its own repository. If the app on the device must be updated, the server also issues an update request, but only after the user stops streaming the app. This triggers the app proxy to perform a background update on the device-resident app.

***Prognostic* Code State Cache (CSC).** The CSC is an edge function maintained in-memory app repository. It can be populated using policies that govern (i) preparation for app streaming, e.g., by pre-fetching in memory those apps that will be used soon based on user behavior, (ii) app refresh on the edge function, and (iii) app replacement to conserve limited resources.

In the Android prototype, for all (uid, app) pairs, there exist executable and static assets in the CSC (e.g., app background images, embedded A-V content in apps, etc.). In order to minimize memory usage on the user's device and the number of requests to app streaming edge function during streaming app execution, we have explored multiple approaches for storing executable. We first used existing single classes.dex file and streamed it at the start of an app, but that needlessly stresses device memory and increases delay, as classes.dex typically span multiple initial packets. Second, we de-compiled apps and create dex file for each class that can be streamed as requests arrive. This is good for device memory but bad for the number of requests. We therefore, maintain multiple dex files for a single app, each including a subset of classes from classes.dex, where the size of each dex file depends on the TCP/IP payload limit, i.e., 65536 bytes, and the order in which these dex files are delivered is derived from the app signature. This requires pre-processing of app classes by edge function (which remains transparent to app developers). Currently, CSC population is performed manually, using the dex2jar [16] and dx tool from Android SDK to extract and repackage dex files which we plan to automate using inotify callbacks on app repository maintained by edge function.

***Invisibility* enabling Device Cell Link (DCL).** The purpose of the DCL protocol is (i) to realize seamless access to app executable and static assets from edge function, and (ii) to reliably stream app executable components. In Android, DCL operates by referring to classes by their qualified names, as seen on the device, using hooks implemented in the Dalvik VM as it gets launched before any loading request is issued by an app. Reliability in streaming is obtained by implementing DCL on top of TCP/IP sockets (vs. using UDP). Operationally, as also shown in Figure 9(b), a connection phase, is followed by a streaming phase in which app classes, assets are streamed to the device running the app.

The AppFlux back-end components span the edge function and the remote cloud hosting app-store service (e.g., Google Play Store). As stated earlier, The key components in edge function-App Store interactions needed for AppFlux are same as AppSachet described next.

3.4.2 AppSachet

AppSachet acts as source of the latest apps and their updates to connected end client devices in similar ways as existing app stores and is placed in the app eco-system as shown in Figure 10. AppSachet sees all requests made for apps and/or their updates to app-stores. Its goal is to leverage redundancy in app traffic and provide benefits to end-users and reductions in last-mile bandwidth use, while operating efficiently within limited resources available to an edge functon. AppSachet operation starts as a simple LRU cache of web responses (from the app stores) which contain the actual binaries of apps and/or updates requested by end clients connected to the AppSachet edge function. If an end user's request cannot be fulfilled by AppSachet i.e., a cache miss is observed then, it proxies the request to remote app stores and saves the response in its local storage. To ensure high hit ratio, AppSachet ranks the seen apps after a pre-definded bootstrapp time, and based on that ranking segments its own cache into two parts. The segment created are either based on app popularity (i.e., in case of p-LRU cache) or cost of caching (in case of c-LRU cache) or simple LRU. AppSachet syncs or pre-fetches popular or cost effective apps and their updates from remote app stores. Thereafter, the popular or most cost effective apps are updated proactively and pre-fetched every hour. When an end client device that supports AppSachet, the device starts by sharing information about installed apps on-device to which an AppSachet responds in form of apps and/or updates available, depending on user-preferences.

For completeness sake, we outline a simpler version of how AppSachet is integrated in the Android app ecosystem, by focusing only on interactions related to app installs and updates. These mechanisms are useful even in the context of the existing app download/install/upgrade model, but their benefits can be further enhanced through systems support for app-streaming, developed in our previous work [57]. AppSachet achieves its goals via the following four components:

1. **AppSachet Cache:** An edge function module maintaining a cache containing apps, updates, and anonymized app-profiles on its local storage.

2. **AppSachet Server:** An edge function server module that services end client devices' request for apps and/or updates, and collects anonymized app-profiles from the connected devices.

3. **AppSachet Sync:** An edge function module pro-actively fetching apps, handling update

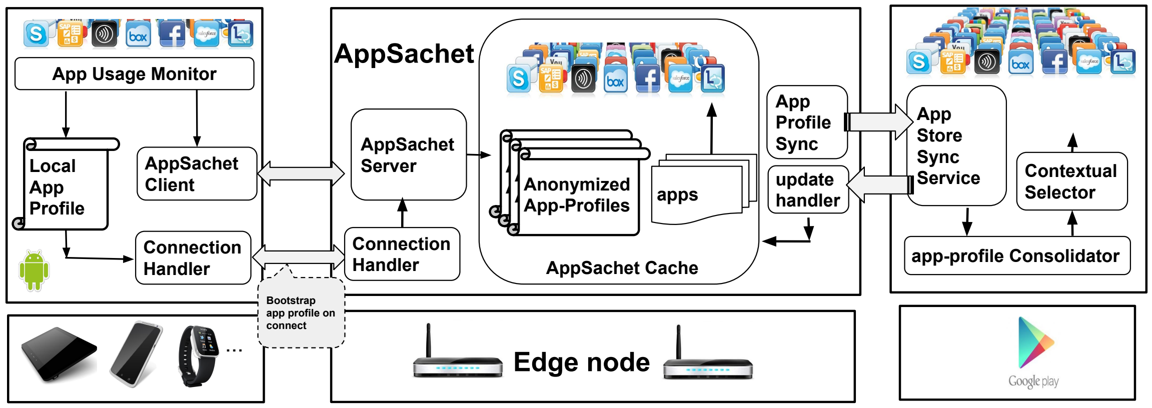


Figure 10: AppSachet design showing different system components, their interactions and their placement in app ecosystem.

notifications from app stores and notifying app stores about the delivered apps and/or updates.

4. **AppSache Client** is a module embedded in the Android app framework that enables handling of app installs and/or updates from an AppSachet server.

3.4.2.1 AppSachet Cache

The AppSachet cache is an edge function module that maintains an indexed repository of app and update binaries fetched from app stores. It houses aggregate app usage information – referred to as *app-profile* – from all connected clients, and a list of delivered apps and/or updates mapped to particular user, used for required app-store notifications. Although the policy used for app cache management can be as simple as an age-based LRU policy, we demonstrate significant gains from targeting the cache management policy to the characteristic of the app traffic. In response, we define two policies – p-LRU and c-LRU – described in greater detail in §3.3.4. The updates of the app cache rely on AppSachet's Sync service.

In addition to apps and their updates, AppSachet also maintains per-app *App Profiles*. An App Profile is simply a relational structure containing the state collected from end user devices on connection. It includes the following user specific persistent information from device: (i) a list of apps, (ii) their versions and (iii) usage patterns of installed on end user device. It also contains session specific device configuration, e.g., current IP address of the device needed to deliver an app or update, and the current App Sachet user preferences indicating how user wants his device to interact with AppSachet. For instance, the preferences can indicate whether a user wants to update all available app updates or to disable updating specific app, of is a user wants to see new contextual apps available for installation, etc. An app-profile is exchanged during the bootstrapping when a device first connects to an AppSachet.

App-profiles are also kept with edge function in another cache instance. The rationale behind keeping a cache vs. a persistent copy of app profiles is first based on the limited amount of storage, and the fact that app-profiles are synced with app-stores anyway. Second, considering the predictability of human movement, i.e., we often go the same places at particular times, e.g. office, coffee shop, etc., creates opportunities for applying proactive and predictive caching algorithms.

Note, however, that sharing this information about a device poses a potential privacy threat; it is avoided by sharing only anonymized app profiles with edge function. The anonymization of app profiles is designed to be carried out on the device, in the app usage monitor, vs. the edge functions, to prevent privacy concerns. Another concerns is mismatch in app version installed on device and the one known by backend app stores due to edge function based updates. For the current prototype, it is a non issue because of the way AppSachet fetches apps and their updates on behalf of an end user effectively syncing the current version of app on device and known by app stores. But we posit that a delegation of authorization from end user to edge function could be used in real-world deployments.

3.4.2.2 AppSachet Server

AppSache server carries out interaction with a device. It is responsible for bootstrapping device-edge function interaction on connection by presenting a valid certificate which established that edge function as as valid provider of apps and updates. Another choice is to have remote app stores involved during the bootstrapping process but that leads to longer bootstrapp process as the device and edge function have to reach out to app stores, which then can issue a common token which can be used to verify identity of an edge function. The server interacts with the cache of apps and shared app profiles, and updates and considers user preferences, e.g., to create a tailored response for the device.

Actual App Delivery from an edge function is facilitated by Android Debug Bridge (ADB) over Wi-Fi to connect to the device and carry out actual app installs and updates when requested by a device resident AppSachet client, an app at a time. The decision to not batch multiple app updates from edge function to end client device is to ensure correctness of updates on a device, and also not to overwhelm the end user device's network with large number of updates.

3.4.2.3 AppSachet Sync

App-Sachet's Sync service is responsible interacting with existing app-stores on behalf of end clients. Its interaction involves (i) fetching apps and/or updates not cached with an edge function and (ii) periodically checking and pre-fetching updates for apps based on p-LRU or c-LRU policy. It supports a pull-based mechanism for update distribution for which AppSachet registers a push notification handler, i.e., **update handler**, listening to push notifications from the app store for apps present in its app cache. When a notification arrives, the AppSachet sync service fetches the updates.

App stores transmit app as full apks to end clients devices but updates are transmitted either (i) as full apks if there exists is a wide gap in version of app installed on device vs. app version that is currently available app store or (ii) as incremental updates [136] which are binary diff of previously installed app apk and the current version of apk submitted by developer at app store. AppSachet supports incremental updates to end clients and also handles incremental updates for its own cache. To ensure correctness of incremental updates, app cache follows a 2 phase commit approach i.e., it commits an update to the app cache only when there are no current users installing the app or its update to avoid misalignment of app versions, but once committed, the update is immediately available to edge function connected devices.

A push-based approach to app cache updates, allowing app stores to dynamically push apps or their updates to a device, could leverage global context, e.g., trending apps, important updates, etc. However, given that our current implementation is limited by the existing unoffical Google Play API, AppSachets are restricted to a pull-based approach explicitly requesting apps and updates from the store.

The Sync component is also responsible for aggregating and propagating to the app store notifications about delivery of an app or an update. These notifications are sent asynchronously to app stores to avoid causing slowdowns in AppSachet-end user device interaction but still ensuring consistency in the versions of apps installed on end user device and what is known to remote app-stores. The choice of lazy and asynchronous reporting to remote app stores by edge function ensures that devices are not burdened to communicate with remote app stores. It also avoids making remote interactions between edge function and app stores a bottleneck while edge function updates are ongoing. However, this may be problematic for apps that require payments. We posit that to support paid apps on AppSachets app stores, either this communication would have to be made synchronous or the edge function must be enabled to process payments. We believe there are additional challenges related to authorization and authentication of edge functions, which we plan to explore in our future work.

AppSachet relies on app store-resident functionality to provide the aforementioned callbacks or edge function-initiated sync operations, and leverages app profiles and other information gleaned from edge function usage patterns to guide the distribution of app updates across edge functions, or to otherwise allow app stores to benefit from the presence of edge functions in the end-to-end app ecosystem. Even though this paper has not yet explored challenges concerning the efficient operation of an edge function-app Store interface, we believe that with $\sim$100 apps installed on a average device [3, 2] and an update cycle of 10 days, there are significant opportunities to reduce considerable overhead from app stores. By using edge function based app delivery, congestion is reduced by (i) providing flexibility in scheduling app store interactions and updates, and (ii) by distributing the app and app update delivery load across a number of edge functions, which then can handle per device installs/updates.

3.4.2.4 AppSachet Client

The AppSachet client is implemented in similar ways as described in AppFlux. The AppSachet client also includes a **App Usage Monitor** interfaces with Android's package manager to get the list of installed apps and uses native hooks to app usage APIs [1] to create anonymous app profiles, stored in a separate file on the device's file system. It is run lazily in the background when the device is locked by the end user. The decision to invoke the app usage monitor lazily ensures that (i) mining relevant information doesn't impact user experience when the device is being actively used and (ii) utilizes the period between user locking the device and system's decision to put device in a deep sleep state to minimize its its impact on device's battery usage. The app-profile is anonymized by passing it through a filter to ensure that information shared with edge function is clear of any personal

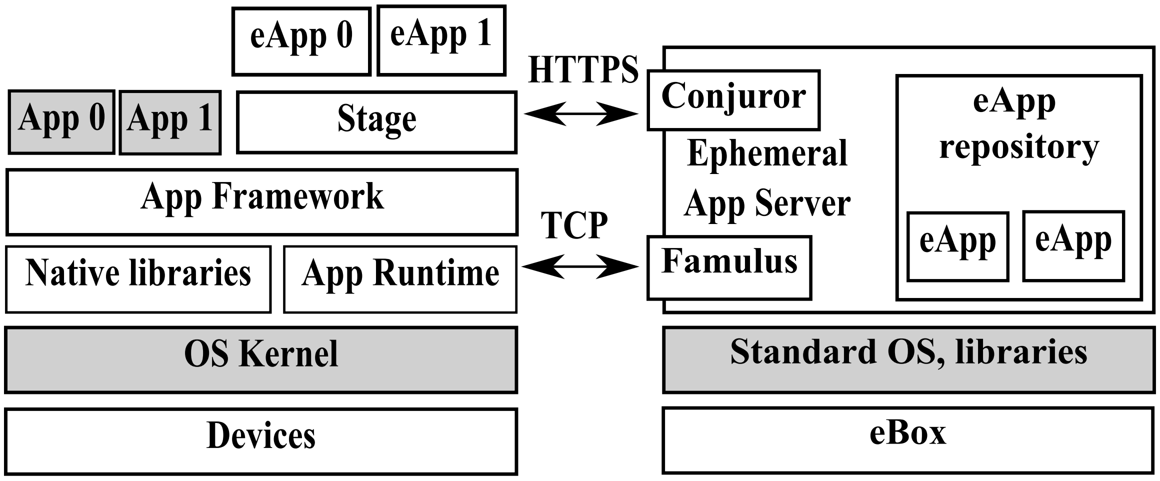


Figure 11: Showing an overview of ephemeral app eco-system and its potential integration with existing app stores.

information. In the current prototype, this simply removes keywords provided by users in their preference, but better anonymization techniques could be deployed for improved privacy guarantees.

3.4.3 Ephemeral Apps

Ephemeral apps extends the app streaming support developed in AppFlux and AppSachet to next level. While the previous work relied on the presence of apps to deliver updates, in this work we assume no-prior information from end user devices to deliver apps to end users. Next, we present the design and implementation details of novel support in Mobile OSes which make *ephemeral apps* possible, which we collectively refer to as **Wandroid** – Mobile OS built on top of Android with support for app streaming and app ephemerality. Figure 11 shows an overview of ephemeral app eco-system and its potential integration with existing app stores.

Wandroid is Android with inherent system level support for (i) app streaming and (ii) app ephemerality. In this section, we delve deeper into implementation details covering the specific modifications in existing Android's subsystems along with the a newly added ephemerality subsystem.

Support for the above mentioned capabilities is designed with the goal of allowing existing Android apps to run in ephemeral mode without any programming effort by developers. This includes support for the following:

Ab-initio app streaming: App streaming support in Wandroid is designed to work seamlessly for existing apps. In Figure 12, the existing Android components which are modified are shaded grey

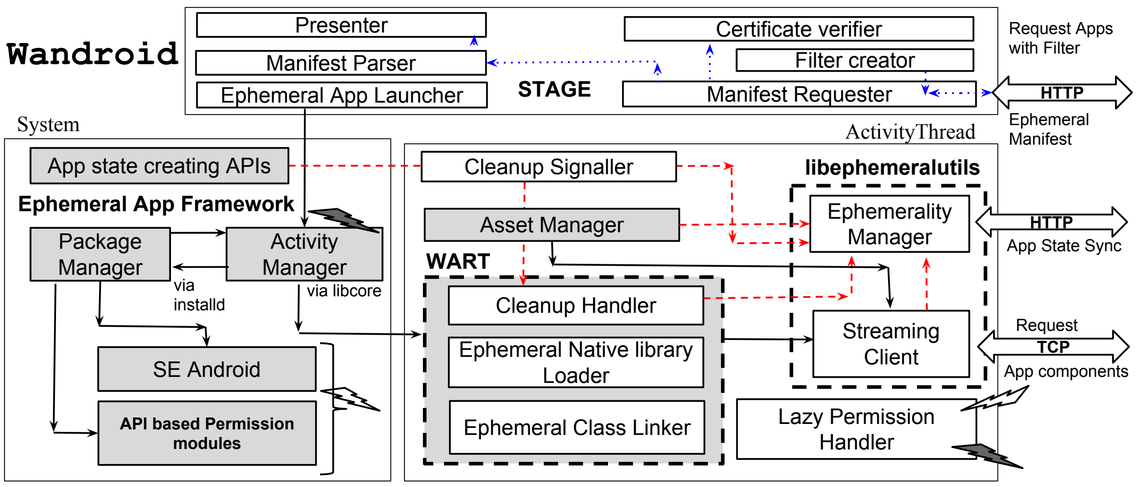


Figure 12: Showing design of app streaming and ephemerality support in Wandroid. Blue dotted lines depict the 'pop up' of ephemeral apps, black solid lines depict path taken by execution of app via streaming, red dashed lines depict handling clean up/sync of app and its state for ephemerality abd block arraows showing interaction with ephemeral app servers.

and newly added components are shown in white. App streaming is designed to operate in two phases (i) *Conjuring phase* and (ii) *Execution phase*. The rationale behind the two phase-design of the ephemeral app life cycle is to ensure faster app discovery by decoupling presentation of an app from its launch and execution described next.

Conjuring phase starts when a user invokes his ephemeral home screen referred to as Stage, as shown with dotted blue lines in Figure 12, on users' device connecting to an ephemeral app enabling edge function. Stage's Manifest Requester module sends the request for available ephemeral apps; on receipt of the response, a certificate Verifier verifies the integrity of the edge function as a valid app provider; If the user trusts them, the Manifest Parser parses the Ephemeral Manifest to start fetching icons of the available apps and asks Presenter to populate those on end user's stage.

Execution phase starts when a user decides to launch an ephemeral app. It starts with call to the a API added in Wandroid as shown in Figure 12 with solid black arrows. Wandroid's Activity Manager starts a new app runtime instance. Before forking an instance of the app runtime WART, the Activity Manager asks the Package Manager to assign a virtual blanket of permissions and to assign ephemeral uid, gid, that belong to a separate range from existing uid ranges used in Android, and are later used to handle permission exceptions to provide lazy permission approvals for ephemeral apps (e.g., if an app requires more than already assigned blanket permissions). To keep app sandboxing guarantees for ephemeral apps while also ensuring availability of storage for caching app components fetched during app streaming, we introduce in Wandroid a new SEAndroid security context and associated policy: ephemeral app domain. During execution, app streaming is designed to work seamlessly for existing apps by intercepting calls to the runtime's class loader, native library loader and the app framework's Asset Manager, and redirecting those to a remote ephemeral app server via a Streaming Client. This ensures that even when the apps are not installed on a device, the running apps are offered the same runtime environment as if they are installed. Concerning state created by apps, which normally is stored in app specific folders on the device's local storage, ephemeral apps are assigned new location which also helps in providing app ephemerality.

Invisible app ephemerality: To track app's system state Wandroid routes all ephemeral app component requests through a central module *libepehemeralutils*. We added hooks in all app relevant framework APIs to capture app private state which is then explicitly logged on-device using an API implemented as part of libepehemeralutils, as show in Figure 12 with dashed red lines. Briefly, the API includes native and JNI interfaces which are used by the Android app framework and by individual system services to log app state when apps use the corresponding APIs. libepehemeralutils' logs are the key to enabling system level ephemerality guarantees. Asynchronously saving, discarding and/or syncing of apps and their states is supported in libepehemrealutils by design, in order to minimize ephemerality-related overheads on the system. The Ephmerality Manager, part of libephemeralutils, allows processing of the logs under different policies, each of which is derived from end user preference about the app state. For instance, if a user indicates that he wants to discard app state then, on app exit, the ephemerality manager accesses the logs to gather app state generated by that app and deletes them followed by removal of the logs from the database. Similarly, if an end user chooses to sync state to an edge function, the Ephmerality Manager sends app state to an edge function before deleting the logs. The logs are created on per app granularity as SQlite tables, with access granted to the corresponding app and system for garbage collection.

Implementation of Wandroid modules responsible for enabling (i) app execution via streaming and (ii) full support for app ephemerality at system level is shown in Figure 12 and is summarized in Table 6.

Stage - Ephemeral Apps Home

Table 6: AppInstant implementation in Android: App streaming, ephemerality and runtime integrity check.

Modules modified	**Functionality**
Framework	
Context	Expose new API for Stage to launch ephemeral API - startActivityEphemeral()
Activity Manager	Handle app slice launch, integrate correctly with activity lifecycle, window Manager, activity stack and Instrumentation, request (uid, gid) lazily, attach seinfo label, handle attaching and binding for app, pass app slices to ART via libcore after setting app classes
Package Manager	Store and provide ephemeral app info, assign (uid, gid) via installd, create app private directories
Asset Manager	Facilitate on-demand streaming of static assets from network, cache them and log using ephemerality manager
Runtime - WART	
Class linker	Facilitate on-demand streaming of only app-specific classes by Hooking DefineClass() JNI used in class loader, loads any JAVA library, cache them and log using ephemerality manager. NOTE: No support for app defined class loader
Native library loader	Facilitate on-demand fetching of native libraries by hooking System.Load() JNI, caches native libs
SEAndroid	
App slice policy	Ensure sandboxing, access to device features using the SEAndroid label, domain, file context
Ephemerality	
APIs instrumented	getExternalFilesDir(), getFilesDir() ,getExternalStoragePublicDirectory(), getExternalDirs(), openFileOutput(), createTempFile(), getSharedPreferences(), getPreferences(), openFileOutput(), openRawResource(), getCacheDir(), getExternalCacheDir(), getWritableDatabase()
Modules Added	**Functionality**
Ephemerality Manager	Provide API for logging, clean up and sync app state using JNI or native interface to libepehemeralutils, and checks their integrity, logs implemented as per app tables in a SQLite's database
Streaming client	Handles app component requests from system and carries out network interactions to fulfil them.
Runtime integrity check	Checks integrity of individual app components at runtime.

In Wandroid, double-clicking the home button, invokes a alternate home screen on the phone, referred to as Stage, vs. viewing the default home screen. Before populating Stage with apps, the integrity of an edge function (or in general a remote ephemeral app server) is established using certificate-based authentication. The 'pop up' of apps is implemented by Stage via an HTTP request for apps to ephemeral app servers with a user defined filter specification. Filter specifies the kind of apps users wants to see on their device, i.e., some user just wants to see apps for near by vending machines only, etc. Filtering of apps ensures that the user screen is not spammed by too many irrelevant apps. By default, all available apps are pushed to end user devices by sending ephemeral manifests in response to Stage request. These manifests are then parsed (by the parser module) to extract the meta-data about available ephemeral apps which contain information such as icon URL, app label and short text description about the app. Additional meta-data (activity name, class implementing the activity, permissions, etc.) needed to launch an app via streaming is also part of ephemeral manifests. Based on the metadata, Stage's presentation module – an implementation of gridfragment – shows ephemeral apps in exactly the same manner as the installed apps are shown on the default home screen. Population of apps results in creation of a file /data/ephemeralapps/applist which contains the list of currently available ephemeral apps and is later used by various Wandroid components to distinguish between installed and ephemeral apps during their execution. On launch, the ephemeral launcher module invokes a newly added path to support app streaming in Wandroid. This path starts from a new Android API – *startActivityEphemeral* – similar to startIntent currently supported in Android. Note, however, that existing apps do not need to use this API, i.e., do not require any changes. The newly added API simply allows Stage to pass the configuration of ephemeral apps required by the app framework and its runtime components to setup its sandbox, runtime environment and launch it.

Wandroid's Ephemeral App Framework

Activity Manager: Wandroid's activity manager is responsible for handling the launch of ephemeral apps and for setting up their permissions. It supports launching activity for ephemeral apps in a way that doesn't interfere with installed apps. The required changes in the ActivityManager include an additional Binder call, support for requesting uid, gids for ephemeral apps at launch time, attaching the correct seinfo label, passing correct arguments to WART via Android's libcore and binding ephemeral app to a newly launched ActivityThread to ensure smooth activity lifecycle transitions, interfacing correctly with the intricate implementation of Android's activityStack and windowManager to ensure smooth transitions while switching between apps.

Package Manager: In addition to handling installed apps, Wandroid's package manager is now also responsible for providing information to the app framework regarding available ephemeral apps, for assigning uids, gids via its interaction with Android's installd, and for creating app specific directories in /data/ephemeralapps/<appname> for their use in caching and/or for ephemeral app private state at launch time. The Package Manager still uses the privileged pipe socket connection to installd to assign Linux level uid, gids, create app specific directories for apk, libraries, etc. Installd's interface and implementation is also modified to support ephemeral apps. Worth noting here is that by setting up ephemeral app directories in /data/ephemeralapps/<appname>, the complexity of supporting ephemerality is reduced. Namely, typically apps do not have access to directory structure on the Android's Linux filesystem, and request specific directories (e.g., media directory, download directory, app private external storage, etc.) via framework APIs. By returning the ephemeral app directory as above results in localization of on-device and/or content state to a single folder, which then can be removed or synced with less complexity, resulting in low overheads.

Asset Manager: Wandroid's asset manager provides supports for on-demand streaming of assets in addition to reading it from a file, directory or compressed apk. Typically, apps do not explicitly request assets in code-line granularity, except essential assets (i.e., icon, AndroidManifest.xml, etc.). This is implemented via the Android framework's Asset Manager, which loads assets on behalf of the app. In addition to making assets available to ephemeral apps in seamless fashion, it caches the streamed assets on device storage at location /data/ephemeralapps/<appname>/<Original PathOf AssetInApk> under control of the Ephemerality Manager. This is also logged to ensure ephemerality guarantee. The current implementation creates a 'temporary' file for the requested assets to preserve the semantics for current apps which require file descriptor of assets. However, the desirable implementation is to create file descriptor using a memory buffer via fmemopem() call not available in bionic[11].

Cleanup Signaller: Cleanup signaller sends a signal to Wandroid runtime to carry out cleanup of an ephemeral app when the app is no longer used. What no longer used means and what actions are to be taken for ephemeral apps is a matter of user preference – e.g., whether a user wants ephemeral app state saved on-device till he moves out of range of an edge function or immediately discarded on switching apps. By default, it is carried out after the app specific activity callbacks onPause(), OnDestroy(). We used the previously unused SIGUSR2 to signal the Wandroid runtime that an app and its state is not needed.

Wandroid Runtime - WART

Currently, Android apps are run as instances of the Wandroid Android runtime (WART), in which we added support for ephemeral apps. Specifically, we added the following features:

Ephemeral class linker: WART loads classes from dex files or OAT files created as a result of ahead of time compilation of classes.dex files of apps. We implemented a new type of class linker which fetches the same dex or OAT files from ephemeral app servers when they are requested via JNI interfaces. With this support, the ephemeral class linker streams only the app-specific classes from ephemeral app servers (vs. reading them from the device's local storage) but still uses the system classes linked and/or required by the app (from device's local storage). This can support caching of the fetched classes if indicated by end user preferences. This also handles the loading of additional JAVA libraries used by an app. However, currently it has a limitation that it cannot use an application defined class loader.

Ephemeral native library loader: Native libraries used in Android apps, i.e., JNI or any other libraries, are loaded by WART which is enhanced to fetch app-specific native libraries from an edge function vs. from the device's local storage at app launch. However, the current implementation saves required native libraries in /data/ephemeralapps/<appname>/lib and opens them using dlopen() as opposed to opting in-memory loading because it isn't allowed by dynamic library loader due to security concerns.

Cleanup handler: A handler of SIGUSR2 added in WART ensures removal of app components (if saved on device's storage) and syncs/discards the app state from the device using ephemerality manager, based on user preference described next, and then, kills itself freeing allocated resources.

Ephemerality Manager

We added a core library *libephemeralutils* in Wandroid which implements an API required for the Ephemerality Manager and for ensuring app ephemerality. The API includes an interface to explicitly log app state, implemented on top of SQlite's C interface and also exposed to the Android framework via JNI. The Ephemerality library also abstracts network operations for streaming of assets and classes for Wandroid's Asset Manager and WART respectively, automatically logging system state created by for app components fetched via streaming. Wandroid's libepehemeralutils logs acts as state that is used by WART for cleaning up and/or syncing app state after when user exits an ephemeral app. The Ephemerality Manager is implemented as a background task which is invoked from WART's cleanup handler when cleanup and/or sync is required.

SEAndroid policy for Ephemeral apps

In order to support similar sandboxing as guranteed by Android for ephemeral apps, we added a new policy to Android's default SEAndroid policies. In addition, ephemeral apps require access to temporary storage to cache app's system and private state. Toward this, we added a new policy for ephemeral apps by introducing a new security label, defining a new security domain derived from untrusted app label (which is used for all apps installed from app stores) to ensure similar security grantees as existing Android. We added a new SE Android file context entry for /data/ephemeralapps/. The policy adds read-write access to /data/ephemeralapps/<appname> for each launched ephemeral apps and keeps all the restrictions of existing untrusted apps as defined by Android. In addition, WART which is run as system process, is assigned read/write access to the base folder to support ephemerality management.

In summary, the additional features and extensions added to the existing version of Android AOSP v5.0.3 result in a new mobile OS – Wandroid, with unique features to permit fast, on-demand execution of streamed apps, and with mechanisms to support app ephemerality, including app state synchronization and cleanup. The implementation of Wandroid are carried out in ways that allow unmodified existing apps to run as ephemeral apps.

3.4.4 AppInstant

AppInstant aims to automatically restructure native apps into app slices that are delivered to end users inside of their messaging apps. Those app slices are then run on end user devices as if they are installed native apps. AppInstant is implemented on top of ephmeral apps support in Android. The additional functionality is creating the functional correct app slices and delivering them as ephemeral apps. Creation of app slices is implemented as offline python scripts that automatically execute apps on an instrumented Android running on a virtual machine. All activities in an app can be extracted by using a tool available with Android SDK to launch all the activities of an app. This results in execution dependency sets for the all different activities in those apps. We then uncompress the apps using 7z and decompile app's classes using dex2jar tool. Then, we rebundle and recompile identified app components creating app slices. Additionally, we extract app slice manifests in similar ways as ephemeral app manifests but at a finer granularity. Figure 26 shows the high level overview of AppInstant. On the app slice server side, part of the chatbot server, app slices are created apriori and are presented to end users who can choose to launch them at which point they are delivered to end user devices. On the end user device side, we add support for new mechanisms in Android to enable app slice streaming and fine-grained runtime integrity checking, to make it possible to present

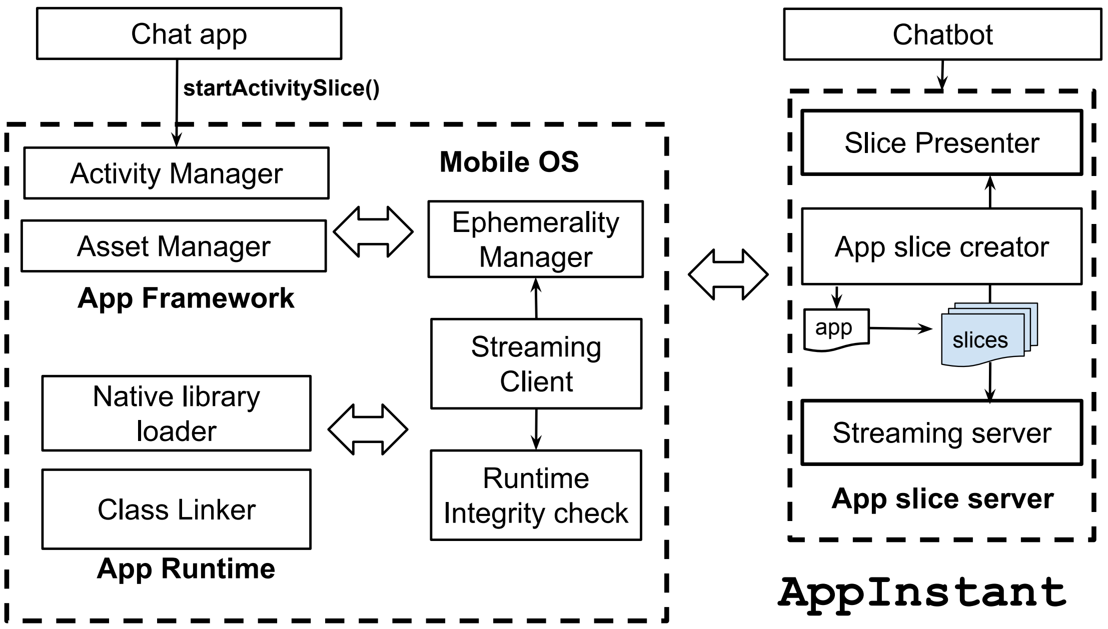


Figure 13: High level components of AppInstant, including the app slice server and the device-side support.

and run those app slices on end user devices in a secure manner. These mechanisms are designed to provide the same experience, security, and performance as native apps. Further, we observed the need to keep end user devices clear of app slices or any on-device state they create during execution.

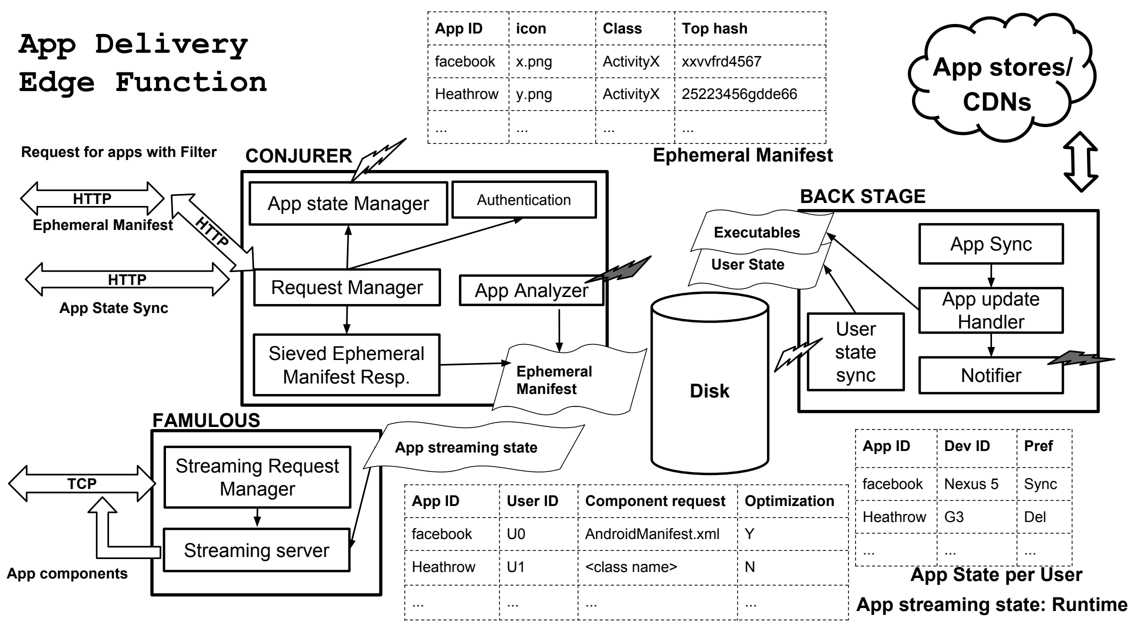
The implementation of AppInstant's device side modules responsible for enabling app slice delivery and execution via streaming, and support for app ephemerality, are summarized in Table 6 and were discussed in previous section.

3.4.5 Enabling edge function

The edge function that supports our vision of app delivery consists of three subsystems named after magic shows as shown in Figure 14:

It is responsible for responsible for making ephemeral apps available to end users. The goals of the ephemeral app service are: (i) maintaining an up-to-date repository of apps, (ii) facilitating 'appearance' of ephemeral apps on end user devices, (iii) handling streaming of app components during execution, and (iv) multi-tenant app state management.

BackStage: refers to a repository of uncompressed apps and their associated states available for streaming of ephemeral apps. It consists of four modules as shown in Figure 14: (i) *App sync*, which



App ID	icon	Class	Top hash
facebook	x.png	ActivityX	xxvvfrd4567
Heathrow	y.png	ActivityX	25223456gdde66
...	...	...	...

App ID	Dev ID	Pref
facebook	Nexus 5	Sync
Heathrow	G3	Del
...	...	...

App ID	User ID	Component request	Optimization
facebook	U0	AndroidManifest.xml	Y
Heathrow	U1	<class name>	N
...	...	...	...

Figure 14: Showing design of ephemeral app server along with states maintained by each components and interaction with clients and app stores.

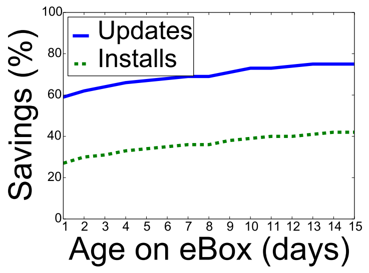
in case of an edge function-based backstage implements an explicit sync/update action to maintain up-to-date versions of apps; (ii) *App update handler*, which maintains the consistency of apps in backstage by applying updates and/or adding new apps to repository; (iii) *Notifier*, which generates a notification for conjurer to update its state for the newly available apps or their updates; and (iv) *Use state sync* which handles sync requests for app specific user state. The role of BackStage can be illustrated via the following example. A user uses facebook's ephemeral app on his tablet at home and he wants to sync the on-device app state to his edge function at a home edge node, e.g., say his preference to aggressively fetch updates. But he wants that preference only for his tablet and not when he uses his phone to use Facebook, and wants this preference to persist next time uses the tablet. Backstage maintains a table per user which links users apps to devices he uses those apps on. The latter can also be used to provide apps components optimized for the user's target device by conjurer. Backstage is implemented as an app storage layer that uses enhanced app metadata. Apps are stored in remote file systems in remote cloud based ephemeral app servers, or on local SSDs, for edge function-based app servers. Linux's notifyd is used to track existing app repository changes and for automatic interactions with Conjurer. App specific pre-processing is implemented using standard Android sdk tools, including the appt tool to extract app metadata housed on the ephemeral app servers' filesystem.

Conjuror: refers to a module that's involved in establishing the integrity of an edge function as a valid app provider at request from Stage and/or at connection time, and in pushing ephemeral manifests to end user devices based on user-provided filter specifications. It consists of six modules as shown in Figure 14: (i) a *Request Manager*: provides CA certificate using the (ii) *Authentication* module and then, handles the requests from Stage for ephemeral app's manifests. It uses the (iii) *Sieved ephemeral manifest responder* to create a list of app manifests based on end user provided filters, which is sent back to Stage. The request manager also interfaces with (iv) *App state manager* which handles app state sync requests arising from Wandroid's ehemerality manager passing them on to backstage's user state sync module if user chooses to use edge function to sync that state to some central entity. On notification about a new app or update from the Backstage Notifier, (v) *App retriever* fetches the compressed app binary and invokes the (vi) *App analyzer*. The latter uses Android sdk tools to extract app meta-data required as part of ephemeral manifest table maintained by Conjurer. App retrieve is again asked to partially uncompress the app components referenced in an app's ephemeral manifest in its own cache to provide faster launch.

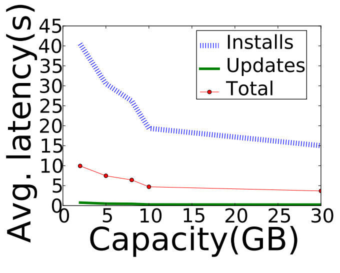
Famulus: refers to a module which actually handles on-demand request for app components; It implements the logic to translate app-specific URIs to uncompressed entities on its Backstage It consists of three modules as shown in Figure 14: (i) *Streaming request manager* which is a Wandroid facing module handling requests from different Wandroid's components for a single or multiple apps; (ii) *Reorder buffer* which maintains the current streaming sessions and handles reordering of requests to optimize for multi-tenant operations and (iii) *Streaming server* which responds back with stream of app components for app execution via streaming.

Creation of app slices is implemented as offline python scripts that automatically execute apps on an instrumented Android running on a virtual machine. All activities in an app can be extracted by using a tool available with Android SDK to launch all the activities of an app. This results in execution dependency sets for the all different activities in those apps. We then uncompress the apps using 7z and decompile app's classes using dex2jar tool. Then, we rebundle and recompile identified app components creating app slices. Additionally, we extract app slice manifests.

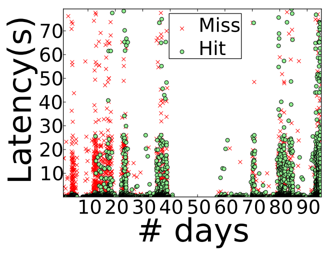
Conjuror and Famulous are implemented as an HTTP and TCP servers servicing requests from



(a)



(b)



(c)

Figure 15: (a) Traffic reduction in the last mile due to apps and their updates with the age of app with edge function i.e., app is stored for those many days after it has been seen for the first time; (b) Average response latency in fetching an app with changing storage capacity: On a cache hit, app is fetched from an edge function else from a remote cloud based app store of different capacity; (c) Latency variation with age of apps/updates as number of days with 10GB of LRU cache.

end-user devices for app metadata and app components, respectively. TCP is chosen for Famulous app streaming because (i) HTTP-based streaming of components incurs high overhead for app components and (ii) reliability in streaming is obtained by implementing Famulus on top of TCP/IP sockets (vs. using UDP). Two versions are currently implemented, using node.js and python, to support easy porting to the many platforms that could act as ephemeral app servers – e.g., edge function, or Amazon EC2 instances.

3.5 Evaluations

In this section, we present the evaluation of the four developed systems carried out as part of this research.

3.5.1 AppFlux

The goal of AppFlux was reduction in the last mile traffic while highlighting that update delivery using streaming from the edge doesn't affect app performance.

3.5.1.1 Traffic Reduction in the Last Mile

Figure 15(a) shows the estimated reduction in traffic if the app installs and updates are cached with edge function for 1 to 15 days. Our analysis of Internet traffic shows the use of edge functions to result in potentially saving up to 70% of the last mile traffic due to app updates and up to 42% of traffic from app installs. We see such huge benefits because when two or more devices near the same edge function update or install the same app, this leads to duplicate requests from all those devices to go to Google Play Store via the Internet. Those apps are then delivered directly or by Google's CDN, but in both cases this leads to repeated delivery of the same apps binaries to different devices in the last mile which holds true for app updates as well. With an edge function, redundant last-mile traffic is eliminated by re-using the single update or install already present with the edge function. This clearly shows that an edge function-based app cache can significantly reduce traffic in the last mile of the Internet.

3.5.1.2 Faster App Delivery

We analyze the impact of edge function on app delivery by considering edge function as a 'nearby' addition to existing CDNs. We simulate an edge function and a single nearby CDN POP (point of presence) using the open sourced Globule CDN implementation [131] deployed on our OpenStack-based cloud infrastructure. Our setup represents the best case for any CDN node because it is present within the same ISP (i.e., Georgia Tech's network) compared to CDN which are situated near ISP's nodes. We use captured app traffic traces to generate a representative app traffic workload. Figure 15(a) shows an achieved cache hit rate of 93% with 10GBs of storage made available to edge function. The left part of the figure shows the cold start of the edge function cache, where subsequent requests to the same app result in cache hits and Figure 15(b) shows the average download latency from an edge function with a 10GB of cache. In case of hit, the app is fetched from an edge function and an app is fetched from a remote cloud based apps store in case of a miss. The app install and update times can be significantly reduced in case of cache hit as can be seen from red and green dots in Figure 15(c). From Figure 15(b), we can see that the average install and update times can be reduced by a factor of 2x, i.e., from 40.29 to 19.362 secs., when the edge function cache size varies from 2GB to 30GB. From these results, we see that a 10GB cache can absorb most redundancy in the last mile due to app traffic. Important points to note here are (i) AppFlux does not compete with the CDN, but complements its operations. (ii) We must consider relative improvements in bandwidth and latency measured as opposed to the absolute values measured in our simulated experiments. An edge function based cache with reasonable storage, i.e., 10GB, can provide up to 2x faster app delivery than CDNs.

3.5.1.3 Streaming Improves App Performance

Experiments are run on a Nexus 5 phone with Android (CyanogenMod 11.2 $\sim$ Android KitKat). The edge function is run on a Core2Duo machine housing apps in its local storage and connected to a Linksys wrt 1900ac router via a Gigabit port. A local wireless LAN using only the 5GHz frequency band (with up to 1.3 Gbps capacity) provides 802.11ac WiFi connections for app streaming. The phone is placed near the router for experiments, with only two entities connected on this network during experiments, i.e., the phone and the edge function. Interactions between the edge function and the app store are emulated by downloading 4,969 apps from the Google Play Store using its unofficial Python API. Figure 16 depicts the time spent in loading app components during execution. The data shown is the median of five measurements, resulting from a total execution time of 10 secs. after app startup, for the Top 100 free apps in each category available from the Google Play Store. Apps are run using the Monkey runner available as part of the Android app ecosystem for app start up, with a fixed numbers of random events (20 UI events with a spacing of 500 ms) mimicking a real use of an app by an end user. We exclude the first run when device-specific optimization is applied by the existing Android dexopt tool, for apps installed for the first time.

Figure 16(a) and Figure 16(b) show the CDF of the time spent in loading app-specific classes and assets, respectively. As seen in Figure 16(a), load times for app-specific classes vary from a few milliseconds to slightly more than 2 seconds, for the 99th percentile of apps. Similarly, Figure 16(b) shows the time taken to load app-specific static assets varying from a few milliseconds to 2 seconds for the 99th percentile of 10 secs. The measurements in Figure 16(c) suggest that the time taken to load system classes, for the 99th percentile, is around 5 secs. These measurements support our assumption about the impact of app loading on app performance. This also suggests that these (surprising) results are primarily due to (i) Android's dex file format, which packs all app classes into a single file to reduce the app's storage footprint, (ii) Android's class look-up method, which uses a hashmap and utilizes a memory mapped dex file to load classes, and (iii) finally, the effective flash memory access bandwidth available to the class loader. The consequent technical approach chosen for app streaming addresses these issues by (i) eliminating the need for compression in the dex file format, by keeping uncompressed app classes and static assets in the memory-rich edge function,

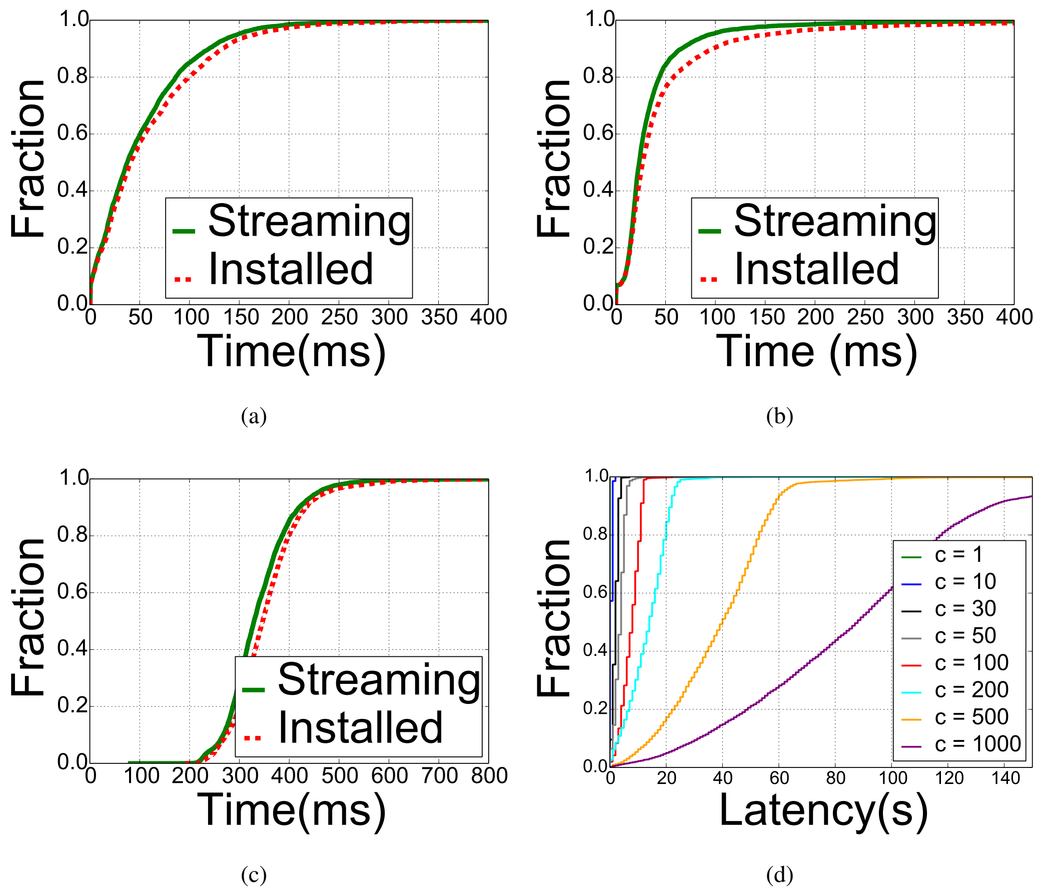


Figure 16: App loading performance comparison (streaming vs. installed) (a) showing CDF of time taken to load app specific classes; (b) showing CDF of time taken to load static assets of app; (c) showing CDF of time taken to load system classes; (d)showing CDFs of latency in total response time per request varying with number of concurrent clients (c).

thus trading storage capacity in edge function for performance, (ii) shifting class lookup overheads to edge functions, by modifying the existing Dalvik class loading mechanisms, and (iii) leveraging the improved access bandwidths of future WiFi networks. A positive additional side effect of improving app class loading is a reduction in system class load times, shown in Figure 16(c), also bolstering app performance. Using streaming apps and under ideal network conditions, app loading performance can be improved by up to 20%.

3.5.1.4 App Streaming Server - Multi-tenancy

For edge function micro-benchmarks, we use a powerful (Intel core i7, 16 GB DDR3 RAM) and 802.11ac capable laptop to simulate concurrent clients connected on the same network link. To show that a single edge function can simultaneously serve a modest number of end user clients, with no or little degradation in app streaming performance, we measure the variation in latency of app streaming with different numbers of concurrent clients, averaged over a fixed number of requests per client (set to 20 per client – because most apps end up fetching 20 chunks of components, including classes and assets). The concurrent client load are simulated Node.js processes, and the bandwidth achieved is limited to 270 Mbps, the latter because of a limitation in the laptop's Wifi adapter that supports a maximum of 8x40 Mbps simultaneous connections on 5GHz. Conservatively using full dex file response payloads for every request (the actual AppFlux implementation does not have such large responses), Figure 16(d) shows that our current (un-optimized) edge function implementation can handle 100 concurrent clients with reasonable latency. Beyond that number, we observe the bottleneck to shift from network to storage I/O on the edge nofr, which leads to degradation in response time, as shown in the figure. This suggests the need for careful edge function implementation and optimization for client multi-tenancy, which we plan to undertake in the future. Nevertheless, the micro-benchmark highlights that even with an un-optimized implementation, an edge function can handle app streaming for a reasonable number of clients, thus establishing the viability of edge function-based app streaming.

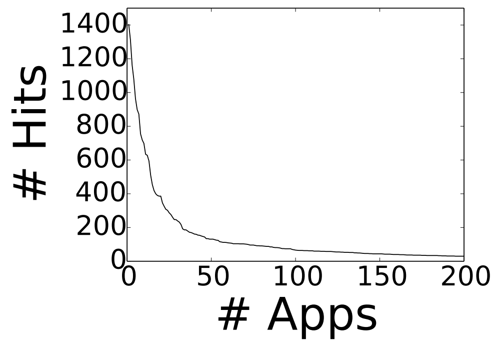
3.5.2 AppSachet

3.5.2.1 Experimental testbed

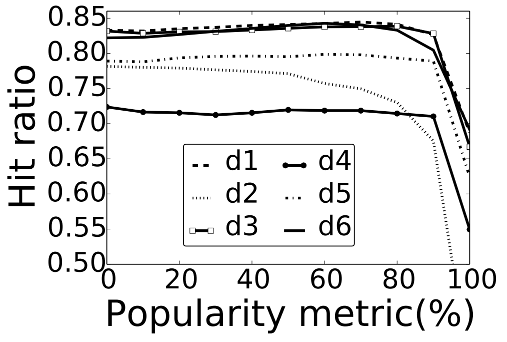
App traffic mesurements are obtained from a network tap that has the capability of logging all traffic flowing in and out of our institution. We used offline analysis to filter the data after logging. The AppSachet is deployed on an edge function run on a Core2Duo machine housing apps in its local storage and connected to a Linksys wrt 1900ac router via a Gigabit port. We generate a representative app traffic workload for our experiments using captured app traffic. The AppSachet client is prototyped using a Nexus 5 phone running Android (CyanogenMod 11.2 $\sim$ Android KitKat).

3.5.2.2 Cacheability of app traffic

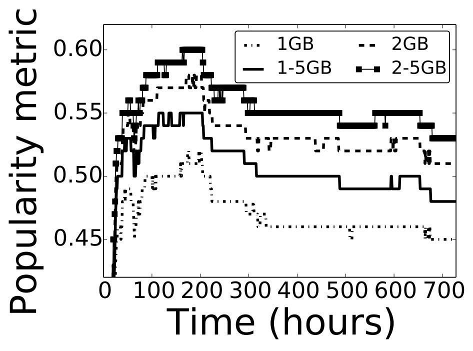
Figure 17(a) shows the number of times a particular app or its update in accessed from the measured app traffic. The most popular app was accessed 1403 times and then access frequency decreases exponentially. Specifically, the 100th app was accessed only 68 times, showing a clear long-tail distribution of apps and their updates. To gain insights into finer temporal cache characteristics of app



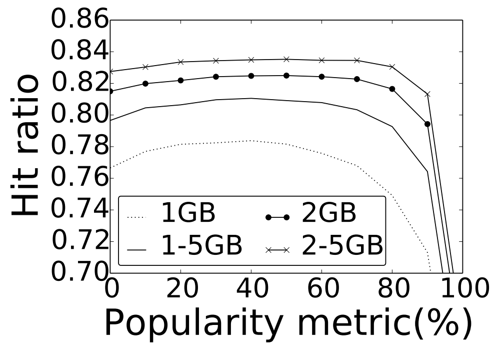
(a)



(b)



(c)



(d)

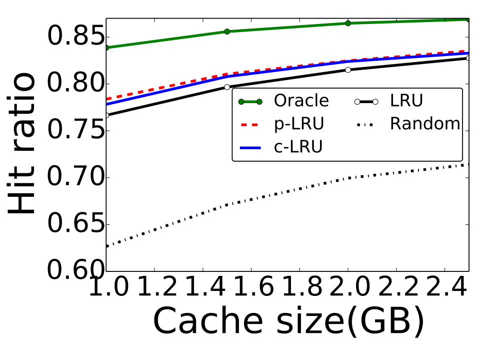
Figure 17: For the measured app traffic showing (a) the number of times each app is accessed in the complete dataset; (b) temporal caching characteristics of app traffic workload by dividing observed traffic in 6 small periods; (c) number of popular apps vs. other apps accessed from the cache per hour; (d) p-LRU cache performance with different size of edge function based cache.

traffic, we divided the complete dataset into 6 equal smaller periods – where each line in Figure 17(b) d_i corresponds to a different period – and found that the caching characteristics persist for small periods as well as for the overall traffic trace.

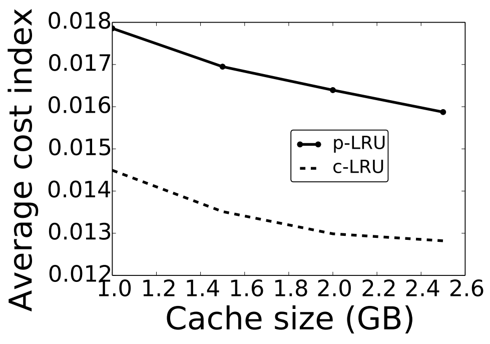
Going a step further, we analyse the observed app traffic on a per-hour basis to capture local popularity of apps on an edge function based cache. Figure 17(c) shows that for every hour, 40%-60% of apps are accessed from what we call the local popular app cache. We notice that every hour, the local popularity of apps on an edge function changes, requiring hourly updates to keep the app cache clear of outdated apps, also seen from the pattern in the Figure 17(d). *These results establish that traffic due to apps and their updates is suitable for caching on an hourly basis and provided justification for our rationale behind the design of the p-LRU and c-LRU caching policies.*

Table 7: For the measured app traffic, showing comparisons of cache policies with varying cache sizes.

	Cache Size	1 GB	1.5 GB	2 GB	2.5 GB
	Oracle	0.8386	0.8558	0.8647	0.8688
Cache Policies	p-LRU	0.7837	0.8105	0.8247	0.8352
	c-LRU	0.7782	0.8078	0.824	0.8328
	LRU	0.7665	0.7965	0.8149	0.8274
	Random	0.6266	0.671	0.6994	0.714



(a)



(b)

Figure 18: For the measured app traffic showing (a) the comparison of caching policies; (b) the average cost index observed (i.e., by all the apps stored on edge function at a time) when using p-LRU and c-LRU caching policy while varying cache size.

3.5.2.3 p-LRU and c-LRU Cache Performance

We compared the proposed p-LRU and c-LRU with a number of popular cache policies, i.e., LRU, Random and Belady's optimal eviction policy. The experimental results, summarized in Figure 18(a) and Table 7 are obtained using edge functions with up to 2.5GB of cache storage.

It is clear that p-LRU outperforms all other policies and is closest to the optimal cache closely followed by c-LRU policy. Figure 17(d) shows the overall performance of p-LRU cache for varying sizes of app cache which shows that the best ratio is obtained when the cache size for popularity metric is between 40%-60% which drops drastically after 80%. This also shows why one segment must be assigned as a LRU cache, i.e., LRU also plays a very important role in maintaining a high cache ratio whereas the popularity metric or cost index ensures that the popular apps or apps with high cost index are always cached, even on their first access. *We conclude that efficient use of the capabilities of upcoming edge cloud platforms (e.g., for caching) would require defining new application specific metrics (e.g., popularity, cost) and/or implement new mechanisms (e.g., p-LRU,* *c-LRU).*

3.5.2.4 Storage Requirements for edge function

Figure 17(d) shows the variation of cache hit ratio of the p-LRU cache with increasing cache size. Using a p-LRU cache with a capacity of 2.5 GB results in the highest hit ratio; this is also closest to the optimal Belady's algorithm shown as Oracle. We conclude that with 2.5 GB of additional storage at edge nodes and a p-LRU cache, AppSachet achieves a 83% hit ratio. Figure 18(b) shows the average cost index observed, i.e., average of cost indexes of all apps stored with the edge function, updated hourly, while running through the complete workload and using p-LRU and c-LRU. As apparent, c-LRU beats p-LRU consistently in terms of lower cost index and hence, lower cost of caching resulting in higher benefits, while still slightly sacrificing hit ratio as seen from Table 7 This highlight a trade off in cache performance vs. cost particularly for edge functions with less available storage due to cost constraints. *Generally, for edge services (e.g., AppSachet) deployed on edge cloud platforms with resource constraints (e.g., storage capacity), designing mechanisms (e.g., c-LRU policy) must consider other factors (e.g., cost) vs. just performance (e.g., hit ratio).*

3.5.2.5 Cost benefit analysis of deploying AppSachets edge function

Without real world deployments of edge function and in the absence of any real cost models for edge function revenue, we base our cost-benefit analysis on the retail cost of SSD storage and the benefit of the latest pricing information about content delivery networks prices. Simply put, the benefit from an AppSachet is directly proportional to the reduction in volume of traffic served by an edge function. Consider the following:

1. There is a wide range on prices offered by CDNs[12], e.g., typically \$0.01 per GB to \$0.05 per GB depending on the volume of traffic.
2. Additional storage cost of 2.5 GB flash storage varies from \$3-\$20 based on its quality. Assuming that we also add 2GB DDR3 RAM as well to the edge function, which costs anywhere from \$10-\$20, this would result in a maximum increase of \$40 in edge function cost.
3. Based on the size of app installs/updates, the total amount of bytes served by an app store are $\sim$2.6TB. With a 83% hit ratio shown to be achieved by p-LRU cache would serve $\sim$2TB from

edge functions.

Conservatively, using \$0.01 per GB, an edge function can save (\$0.01 x 2000 = \$20) in three months, i.e., an edge function would be able to recover the additional cost of storage within 3 months of its deployment. Even if we consider the additional RAM as a cost increase in deploying edge nodes, it will be recovered in the first 6 months of edge function deployment. *From this, we want to highlight the value proposition of edge cloud based services (e.g., AppSachet) in terms of reduced operational costs for cloud based services.*

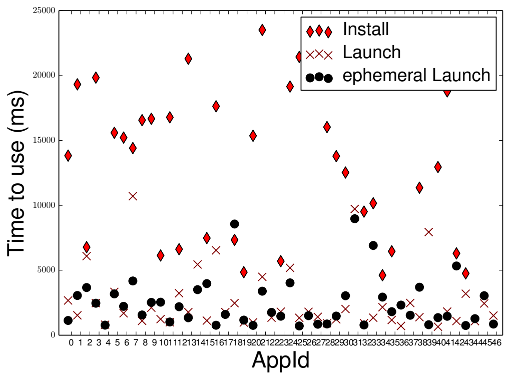
3.5.3 Ephemeral apps

Experimental Setup: We used a Nexus 5 phone running Open Source Android v5.0.1.3 for device side evaluation. The phone is connected to edge function running on Linksys WRT1900ac 802.11ac Wi-Fi router running OpenWRT. The same router with an attached SSD of 20GB is used to prototype the edge function based app streaming server.

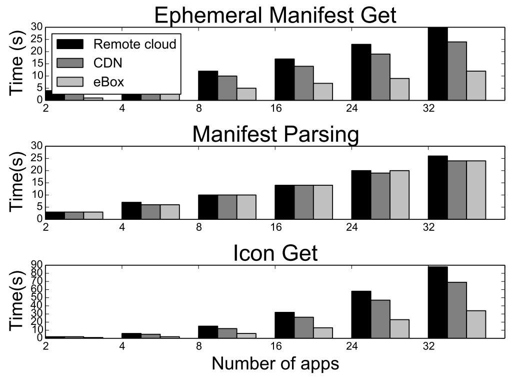
Feasibility: We report that we have been able to run native existing apps as 'Ephemeral apps' via app streaming and app ephemerality on Android without requiring any changes in those apps.

Time to use: Figure 19(a) shows the lower bounds of time to to start using an app when installing it from app store vs. using ephemeral apps. In addition, Figure 19 shows upto 10x reduction in time to use compared to installation and launch time of top 40 apps which include the top app in each category downloaded from Google Play Store. Also, important to note is that, launching app via app streaming doesn't lead to delay in app launch time and hence, leading us to believe responsiveness is comparable to native installed apps.

Figure 19(b) shows the variation of delay in populating or 'popping up' apps on end users new home screen - Stage. The measurement is broken in three parts, time take to get ephemeral manifests, parsing of those manifests and finally fetching icons for ephemeral apps varying with number of apps to be populated. We observed that in less than a minute the user-screen can be populated with 24 newly available app icons, i.e., the maximum number of app icons that fit on a typical phone's home screen. Also, we observe that edge function-based servers can do the same job in under 30 seconds compared to using CDNs or remote clouds servers. Additional optimizations like icon caching can further reduce this delay. Also, it is worth noting that by making it explicit for users to invoke Stage



(a)



(b)

Figure 19: (a) Comparing time to use of ephemeral apps made available from a nearby edge function vs. lower bounds of install and launch time for top 40 apps. (b) Stage performance: (i) Delay in popping up ephemeral apps varying number with number of apps from edge function, CDNs Remote cloud based ephemeral app server.

Table 8: Showing number of ephemeral logs (classes and static assets requested) during launch of app slices.

Footprint	**Minimum**	**Maximum**	**Median**
Number of Classes	75	2070	1090.5
Size of Classes (KB)	48	2030	970.0
Number of Assets	1	353	31.5
Size of Asset (KB)	5.5	18829	358.6
Number of Native libraries	0	45	1
Size of Native libraries (KB)	0	29581	17

and because apps are downloaded from local network when ephemeral app servers are deployed using edge functions, Stage doesn't burden end user devices with any background overheads.

Ephemerality overheads

Overhead due to ephemerality arises from two basic operations: (i) logging, which includes writing to the ephemeral database implicitly during streaming of app components, or explicitly via JNI when an app tries to create and/or modify its on-device state, and (ii) processing those logs by reading from the database and removing the state by either modifying or deleting files. So, fundamentally the overheads are accesses to the SQLite3 database, JNI overheads, and the amount of state created by an app slice on devices' local storage. It is worth noting that by keeping the implementation inside of the mobile OS, i.e., by modifying Android framework APIs that are primarily responsible for granting and exposing local storage to apps, allows us to provide access to the same local storage, but at the same time, exposes a different location to app slices for their use.

This ensures that support for ephemerality does not incur any additional overheads in the critical path of app slice execution. However, for app slices, additional system state is created and maintained while streaming app components. Concerning that, Table 8 shows the number of requests and the total bytes transferred in form of classes, assets, native libraries or any non-asset app component, during the launch of the 45 apps used in evaluation. We claim that the overhead introduced to support ephemerality is negligible, evident from the fact that launching apps from edge functions can even be faster than from phone's local storage, as shown in Figure Figure 19(a), subject to the quality of connectivity between a device and an edge function. In favor of brevity, we omit two set of results from this paper (i) benchmarking results of JNI overheads and SQLite3 performance and (ii) amount of on-device state of created by apps on local storage, which has been studied extensively in previous research [104, 105, 101, 103] around the impact of storage access pattern on app performance.

Next, we discuss another approach to app delivery that addresses the need of emerging chatbot based user interactions with apps.

3.5.4 AppInstant

Workload: We perform more detailed evaluation of app slicing and responsiveness with a small subset of 45 apps listed as top apps on Google Play store. We chose those 45 apps by (i) selecting 40 popular apps, also ensuring that we include the top app in each category, and (ii) including 5 apps that are known to pressure specific resources on end user devices, also shown in Table 10.

3.5.4.1 App slice creation

We report the feasibility of automatic creation of app slices from Android native apps without any involvement of developers. We created app slices from the top 5000 apps we downloaded from Google Play store and were able to run those slices on end user devices as if they were natively installed. For brevity, we only present the results for the subset of 45 apps. Figure 20(a) shows (i) the total number of activities in the app and (ii) the aggregate size of app slices compared to the total package size and. It shows that the app storage footprint increases with increasing number of activities. The overall increase in the aggregate per app size due to app slices is as expected, as it removes the optimization to reduce the size of app in dex file format which packs all classes into one file. That is offset by only using small number of static resources for each activity. Note that we

Table 9: Experimental setup for AppInstant evaluation.

Setup	Description
Phone	Nexus 5 phone, Wandroid based on AOSP Lollipop v5.0.1.3
edge function	Linksys WRT1900ac 802.11ac Wi-Fi router, OpenWRT, 20GB Samsung SSD via USB 3.0
Simulated CDN	OpenStack Ubuntu 14.04 VM on local cloud test bed: 4 vCPUs, 16GB Memory
Remote Cloud	Amazon EC2 Ubuntu 14.04 instance (m3.xlarge): 4 vCPUs, 15GB Memory
Internet	Gigabit ethernet connected via cable to edge function

Table 10: Apps used in AppIntant evaluation with rationale in choosing them.

App	Category	Rationale
40 apps	Topmost free apps from each category	Evaluation spans different app types.
Facebook (S)	Social	CPU, storage intensive
Chrome (W)	Web browser	Network, Memory intensive
Zedge (P)	Personalization	Network, power intensive
8Ball pool (G)	Game	CPU, power intensive
Guidebook (E)	Apps for events	Network, power intensive

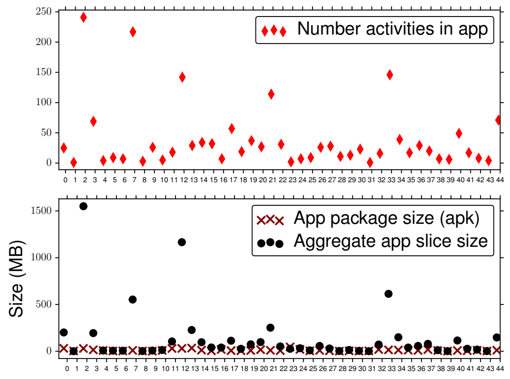
pointed out earlier in Figure 5 that the footprint of typical activities is only fraction of the overall app package. App slicing can be performed a priori, and this can be done for all relevant device form factors, similarly to how YouTube pre-generates versions of videos for different. In addition, app slicing can be dynamically performed, proactively or on-demand. The actual time to perform the app slicing varies based on the app and the number of activities it has, as well as on the AppInstant servers. Using a client-grade machine, we measured that for most apps, slicing was performed in order of few seconds.

To integrate AppInstant, messaging apps simply need to use the new Android API to present and launch app slices from within their chat windows. With the proposed support for app streaming, runtime integrity check and app ephemerality in Android, there is no additional change for them. This substantiates our claim that AppInstant can potentially allow chat bots to leverage the momentum of the native app ecosystem without involvement of app developers and also with minimum effort from messaging apps.

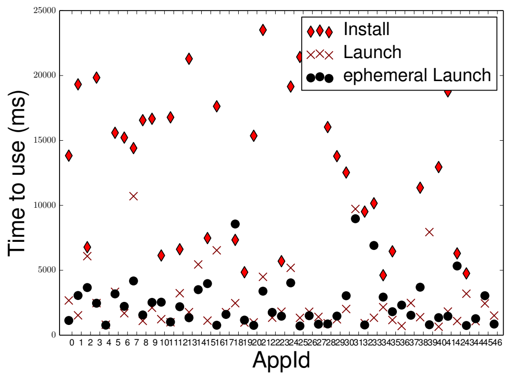
3.5.4.2 App slice performance

Experimental setup: Table 16 lists the details of the experimental setup used in the experiments. The testbed includes a device, and several possible alternatives for an AppInstant server deployment.

Responsiveness. We use launch time as a metric for measuring responsiveness. This is the time



(a)



(b)

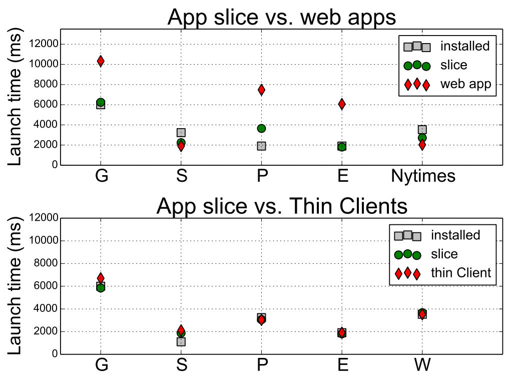
Figure 20: (a) Showing (i) number of activities in apps and (ii) aggregate size of app slices relative to original app package with number of activities marked on top; (b) Comparing lower bounds of installation time (using adb over USB and launching app) and launch of installed apps vs. app slice launch using AppInstant server

taken by an end user to start using an for the app slice for the first time. We believe it is crucial because it represents the time a user needs to spent to try out a recommended app slice. We believe that this is similar to the web page load time metric for web apps based interactions. The lesser time it takes for a user to launch an app slice, the higher the chances that the user will be happy. Further, launching is the heaviest activity in terms of responsiveness as at this time, requiring that almost all of the app slice components be present on the users' device.

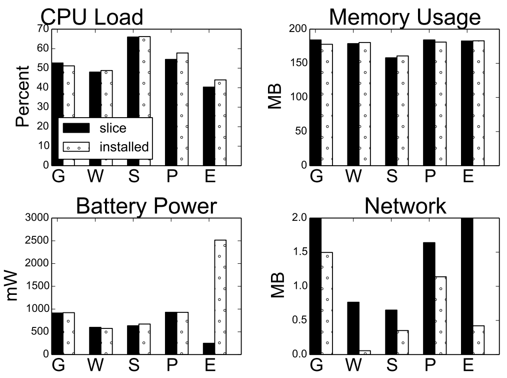
vs. native apps. For native apps, the minimum time required for a user to use an app equals the sum of the app install time and app's first launch time, i.e., ignoring completely the time taken to download an app from an app store over the Internet. In contrast, for AppInstants' app slices, it is simply its launch. We measured it by installing apps on a phone from a local machine, where the apps were downloaded apriori, using Android's debug bridge (ADB) [5]. So, this represents lower bound of the delay in fetching apps from app stores, because it eliminates network delays and uses apps available on hard disk of the machine and installed over USB 3.0, along with launch time of the apps after installation. Figure 20(b) shows the comparison between native apps and native app slices served from a local machine running app instant edge function.

Next, we present our results for 5 resource intensive apps to compare the responsiveness of app slices when compared to other potential technologies that can be used to offer similar functionality – web apps and thin clients.

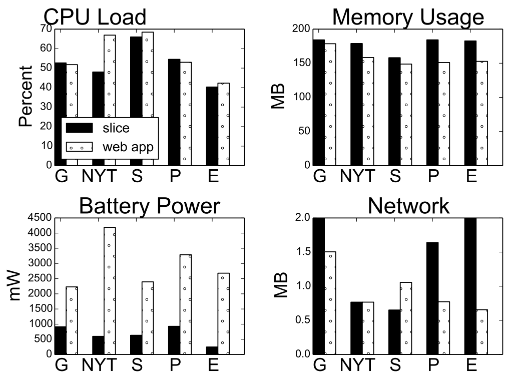
vs. web apps. To compare web apps vs. app slices, we use the 5 apps from the Table 10 because



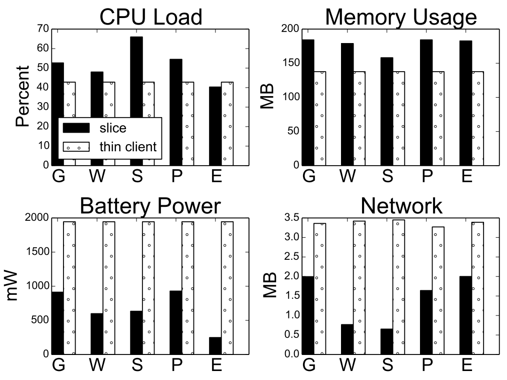
(a)



(b)



(c)



(d)

Figure 21: Comparing (a) launch time observed for (i) app slice using remote cloud v/s web apps and (ii) app slice using edge function vs. thin clients; (b) app slices using edge function vs. native apps, (c) apps slices using remote cloud vs. web apps and (d) app slices using edge function vs. thin client in terms CPU load, memory usage, battery power and network use measured during execution.

both web app and native apps versions are available for those. We compare app slices served from an AppInstant server on remote cloud to web apps, for it is the only fair comparison to serving web apps. We use a developer version of Chromium to access those web apps while controlling the cache behaviour. Figure 29(a) (i) shows the responsiveness in terms of launch time for those apps, not including the browser launch time.

vs. thin clients. Performing comparisons with thin clients raised a number of technical challenges. Unfortunately, there are no standard thin client servers that work for Android Lollipop because they all depend on capturing the frame buffer, but in recent Android versions there is no simple frame buffer [6]. So, we implemented a thin client using the support for secondary screen in Android. These screen captures are then sent to clients from a standard VNC server. We also ran the same apps on Android-x86 running in a VM on a machine connected to a local network – best case for thin client apps. For fair comparison, we used a VM with the same configuration on the same machine to run the AppInstant server. Figure 29(a)(ii) shows the launch time for those apps. Another challenge was to measure app launch in thin clients. The method we used to capture launch time was inspired form calculation of web page speed index calculation [46]. We recorded the thin client session on the device and then, manually subtracted the time between when the app is launched and the screen completely appears. To be fair, we kept the screen recording running while we ran AppInstant as well.

Different deployment models. Further, Figure 20(b)(ii) shows the comparison of launch time when running the AppInstant server from an edge function, CDN, or from a remote cloud. The findings highlight that when served from the edge tier of the infrastructure, app slice can provide responsiveness comparable to native installed apps.

Execution performance. We performed performance experiments with all of the listed apps but present results using the last 5 apps from Table 10, due to limited space and because these apps are known to put pressure on device resources. We carried out our measurements using the diagnostics tool Trepn Profiler, that measures and profiles system resource use, including power consumptions, of applications running on Qualcomm's Snapdragon mobile processors. Nexus 5 uses Snapdragon 800. We compare CPU load, memory consumption and power consumption of installed apps vs. app slices from edge function. During the experiments, we exercise the UI manually for at least 2 minutes in natural ways. For instance, for Facebook, we opened the app, scrolled down, opened a post, went back to wall and updated the wall. Similarly for Chrome, we launched the browser, opened a predefined website, scrolled the webpage, followed a predefined link on that website, opened a second tab, opened another website, then closed the second tab during both runs. We made sure that we follow the same steps and clear any on-device cache of these apps before running experiments on both occasions.

vs. native apps Figure 29(b) shows the raw CPU load, memory usage and battery power consumed. As seen from the graphs, there is no considerable difference in resource consumption for an installed app vs. an app slice from an edge function. There is no impact on power partly because we found that 99.8% of apps ask for internet permission and use the network anyways during their launch and/or execution to fetch ads, updates, content, etc. As a result, streaming app slices does not substantially affect network power, which could be the main concern for app slices (due to WiFi use). However, we want to state that these experiments were done in lab conditions where only the phone was connected to edge function and the performance may be affected by the quality of connection.

vs. web apps Figure 21(c) shows the difference in CPU load, memory usage, power consumption and the network use in terms of total number of bytes transferred when using a web app vs. an app slice. We compared web apps to app slices hosted on the remote cloud – an EC2 instance for it is only fair comparison. As can be seen in the figure there are benefits in terms of reduced launch time and resource power consumption. We attribute these to the number and size of web app components (JavaScript, HTML, CSS etc.) that need to be fetched and the additional processing in creating DOM objects, handling dependencies and rendering those on device, the first three of which are removed when we fetch appropriate native app components via streaming. Further, Figure 21(c) also shows that fetching app components has negligible affect on total network bytes, an observation strengthened by ubiquitous use of Internet to access app content. It seems counter intuitive that more bytes are fetched via the network for app slices vs. web apps, where as memory requirements for app slices are lesser. This is because the fetched components inflate for web apps (DOM objects, render page layout, etc.). For app slices the components can be directly used, resulting in lower battery use.

vs. thin clients Figure 21(d) shows the difference in CPU load, memory usage, power consumption and network use in terms of the total size of data fetched by an end user. As expected, thin clients use the least amount of on-device resources as they only keep at-most two memory buffers (in case of double buffering in clients) of size that is needed to store a RGB image of screen. Despite lower memory and less processing, thin clients consume more power because of its aggressive use of network.

3.6 Research contributions

Overall, this chapter makes the following research contributions:

1. **New caching policies in edge functions:** Derived from real world measurements of app traffic, we developed p-LRU cache and c-LRU cache policies. They outperform other popular cache policies in terms of hit ratio (§3.5.2.3). While p-LRU maximizes hit ratio – 83%, c-LRU

minimizes the cost associated with caching using edge functions eBoxes.

2. **Cost-benefit analysis for edge function:** We show that the cost of deploying AppSachets on eBoxes can be fully recovered by app stores within the first 3 months (§3.5.2.5) of its operation. We estimate the additional cost of deploying AppSachets in terms of the cost of storage required(§3.5.2.4), while benefit is estimated based on the pricing of CDNs.

3. **New concepts proposed for Mobile OSes:**

   - **Ab-initio app streaming:** Design and implementation of app streaming with a notion of *ephemeral apps* for the Mobile OSes. Additionally, we highlight and address the gaps in existing mechanims in Mobile OS i.e., to ensure app integrity especially for streaming apps. We demonstrate feasibility and evaluated it with a concrete realization using Android.
   - **Invisible app ephemerality:** Design and implementation of a new and useful system level ephemerality property. We demonstrate its feasibility using a concrete realization in Android. it is important to note that ephemerality is a weaker property than forensic deniability and therefore can be supported with much lower overheads.

3.7 Discussion

Cellular network. An important question that is left open in this paper is whether AppFlux is relevant for cellular technologies because all of the experiments are run using Wi-Fi. We posit that similar opportunities exist in future cellular technologies using small cells as eBoxes. However, we could not experiment with small cells because of lack of open small cell platforms that we could use carry out experiments as most commercially available small cells are closed platforms (sold by cellular providers) making it close to impossible to insert app streaming functionality on them. Subject to availability of open cellular platforms, we are exploring ways to include evaluation of eBox functionality on cellular technologies.

Deployment Model. Given that eBoxes deployment doesn't exist as yet, there are open questions like who will own the eBoxes – individuals, businesses or will they be part of public infrastructure ? How to securely run app streaming servers on those eBoxes ? How can an end user trust an app from a particular eBox ? Another aspect is mechanisms to manage and ensure DRM of the app repository on eBox. We posit that with existing authentication and authorization methods deployed on eBoxes, this can be addressed theoretically. But it leaves out one concern that is not addressed in existing methods i.e., current authorization and authentication assume a human user which is authenticated or authorizes which is not the case for eBox. We believe that this is an interesting problem which plan to study in details.

Privacy Concerns. Sharing app signature with an eBox may raise privacy concerns as it effectively represents how an app is being used by a particular user. We argue that this would not be more invasive than current support available in apps. Specifically, with recent rollout of app usage API in android[1] which allows developers to track app usage, it seems that sharing app signature is more obfuscated than what is guranteed in latest version of android. However, this aspect certainly needs a detailed evaluation.

On devices executable format.. Further, we plan to push in two directions: (i) Current design of executable file format is driven by the assumption that executables are stored in the local storage of devices but for streaming apps, these assumptions do not hold. We are pursuing exploration of the effects of app layout e.g. dex for android app classes, elf for native code, etc. for streaming apps.

Support in other Mobile OSes. We make an important observation that all modern mobile OSes support similar architecture. All of them have grown taller to include some form of app framework (APIs) to facilitate access to device features, app runtime to sandbox app execution, mechanisms to handle app permissions, support to allow apps to maintain some private on-device state. For this reason, we claim that although AppInstant is designed and evaluated for the Android, similar design will hold for other Mobile OSes.

Energy consumption on devices. With almost all apps requiring network access (98.9% in the apps we used for evaluation) and hence, turning on the network connection, We posit that app streaming can potentially piggyback the tail energy [129], causing minimal affect on overall energy consumption which requires detailed evaluation.

3.8 Chapter Summary

In this chapter, we presented systems demonstrated to realize the new vision of app delivery using edge functions without compromising app performance or requiring changes to apps by developers and/or changes in how end users employ these apps. Our evaluations demonstrated how the use of edge functions can address difficult existing problems of a large scale computing ecosystem such as Android app ecosystem. Further, we demonstrated that the use of edge functions with the new system software support can potentially address non-technical problems such as transition between different models of web service consumption. In summary, we demonstrate with concrete use cases that edge functions can provide value far beyond just latency and bandwidth optimization. It is this potential of edge functions, which if delivered generally to other web services can lead to transformational changes in computing landscape, that informed our nest steps in this thesis.

CHAPTER IV

FROM APP DELIVERY TO GENERIC EDGE FUNCTIONS

Translating advantages similar to those demonstrated for Android app eco-system in previous chapters for other web services requires a more generalized approach. Towards that, in this chapter, we first elaborate on the definition of edge functions and their benefits and then, present the six representative generic edge functions which we identified to model those benefits. We draw on our experience in developing those generic edge functions and those developed for Android app ecosystem to elicit system level requirements and technical challenges to be addressed to the translate the benefits of edge functions to arbitrary web services.

4.1 Edge functions

Earlier in this thesis §1.2.1, we defined edge function as

Any third party service deployed on edge infrastructure that interacts with end client requests on behalf of a back-end service deployed in remote clouds.

However, before we discuss generic edge functions and system level functionalities required to translate their benefits to arbitrary web services, it is important to elaborate on that definition and provide a concretely model of edge functions that can enable reasoning about their benefits more formally.

4.1.1 Modeling Edge functions

Figure 27 shows the proposed model of a generic edge function including the parameters, also enlisted in Table 11, that can be used to formalize their benefits. Concretely, an edge function can be represented as follows:

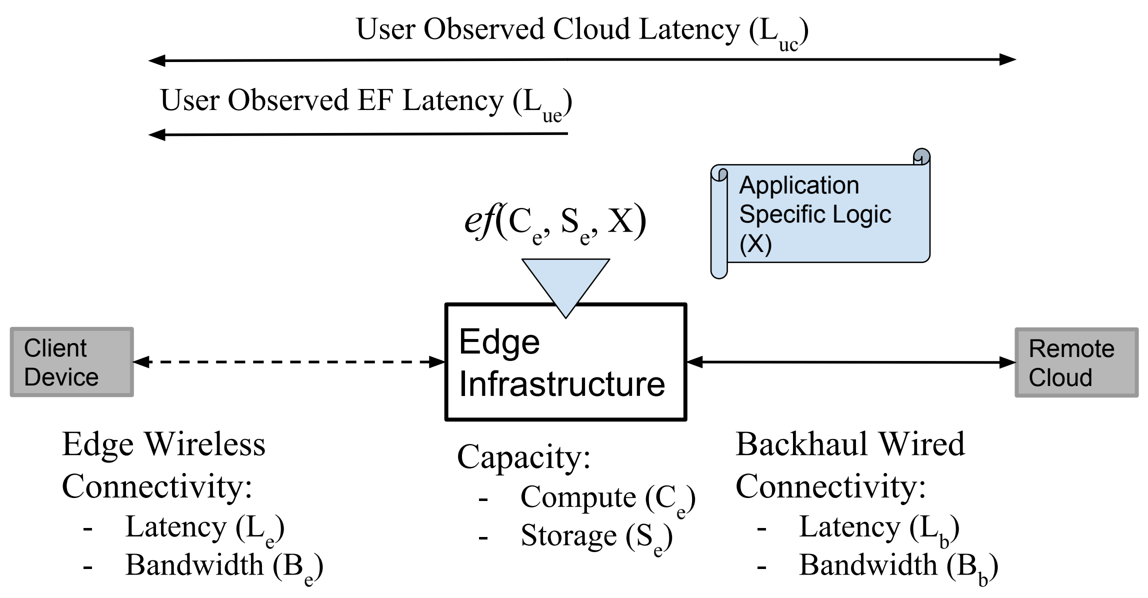


Figure 22: Showing the model of a generic edge function.

Table 11: Parameters in the edge function model to formally describe their benefits.

Parameter	**Description**
L_e	Latency of wireless connection between edge infrastructure and client device
B_e	Bandwidth of wireless connection between edge infrastructure and client device
L_b	Back-haul latency of wired connection between edge infrastructure and remote cloud
B_b	Back-haul bandwidth of wired connection between edge infrastructure and remote cloud
X	Application specific logic embedded in edge functions.
L_{uc}	User observed latency from remote cloud based web service.
L_{ue}	User observed latency from edge function.
B_{sc}	Service observed back-haul bandwidth utilization for remote cloud based web service.
B_{se}	Service observed back-haul bandwidth utilization when using edge function.
C_e	Compute capacity made available to edge function by edge infrastructure
S_e	Storage capacity made available to edge function by edge infrastructure

EF = *ef*(C_e, S_e, *X*)

where,

C_e : Compute capacity made available to edge function by edge infrastructure.

S_e: Storage capacity made available to edge function by edge infrastructure.

X: Application specific logic embedded in edge functions.

4.1.2 Modeling edge function benefits

In this section, we present a formal model to capture the benefits of deploying edge functions i.e., latency and bandwidth. We then show how we can use that model to capture the aggregate benefits and enable reasoning on cost modes for edge computing ecosystem.

Latency benefits

The use of an edge function allows for lower user observable latency end user observed latency for a web service which deploys its edge function. We can formally represent those benefits as described below:

Let

L_{uc} be the access latency observed by an end user when accessing a web service deployed in the remote cloud

L_{ue} be the access latency observed by an end user when accessing a web service from the edge

L_e be the access latency observed by an end user to reach an edge function at the edge

ef(C_e, X) is the time spent in servicing the request at the edge

using the above formulations, we can state the following:

$$L_{ue} = L_e + ef(C_e, X)$$

and ΔL denoting the reduction in access latency observed by user accessing the web service can be derived as follows:

$$\Delta L = L_{uc} - L_{ue}$$

substituting for L_{ue} in terms of end user observable parameters

$$\implies \Delta L = L_{uc} - ef(C_e, X) - L_e$$

substituting for L_{uc} in terms of edge infrastructure observable parameters

$$\implies \Delta L = L_b - ef(C_e, X)$$

From the above expression, it is easy to see that the condition for useful latency benefits is that the access latency from the edge function must be less than latency when accessing from the remote cloud which can be expressed as below:

L_{uc} > L_e + *ef*(C_e, *X*) ... (1)(a)

or

L_b > *ef*(C_e, *X*) ... (1)(b)

Important to note here is that we consider that the latency of edge function to be solely dependent on compute available at the edge infrastructure. One may argue that it may be dependent on storage as well. To that our argument is that, we can keep in memory all the relevant state that is needed for latency sensitive operations at edge infrastructure. This transfers that dependency and hence, bottleneck from storage to compute. Hence, making the latency benefits of edge functions solely a function of compute resources available at the edge infrastructure.

Bandwidth benefits

Similarly, the use of an edge function allows for lower use of back-haul bandwidth for redundant accesses for a web service. Since, the aggregate use of bandwidth can only be observed from either by the web service or an edge function, the formulation is based on their perspective.

Let

'n' be the total number of requests at an edge location.

As a result, a web service observed a bandwidth consumption of B_{sc} due to all its requests.

'r' be the total number of requests that can be handled at that edge location by an edge function denoted by $ef(S_e, X)$ on behalf of web service.

As a result, a web service observed a bandwidth consumption of B_{se} due to all its requests. For simplicity we assume that all requests lead to equal bandwidth usage. So, we can formally represent the benefits as follows:

$B_{se} = B_{sc}(1 - \frac{r}{n})$

where, r is directly proportional to *ef*(S_e, *X*). Omitting the constant, we have

$\implies B_{se} = B_{sc}(1 - \frac{ef(S_e,X)}{n})$

and ΔB denoting the reduction in bandwidth usage can be derived as follows:

$\Delta B = \frac{B_{se} * ef(S_e, X)}{n}$

substituting for edge observable parameters

$\implies \Delta B = \frac{B_e * ef(S_e, X)}{n}$

substituting for web service observable parameters

$\implies \Delta B = \frac{B_b(1 - ef(S_e, X))(ef(S_e, X))}{n}$

Important to note here is that the expression $ef(S_e, X)$ can only take values less than one and greater than zero depending on the storage constraints and the application specific logic to keep or pre-fetch the relevant content at edge function. The expression $\frac{ef(S_e, X)}{n}$ can never be more than 1 because it would mean that an edge function can provide bandwidth benefits for a request that was never made which obviously does not make any sense. A concrete example of this was demonstrated in p-LRU caching policies proposed in earlier chapters which is demonstrated to beat simple LRU cache. From the above expression, we can derive the expression for valid bandwidth benefits as follows:

$\frac{ef(S_e, X)}{n} < 1$ ... (2)

There are two important points to be noted in this discussion. First, we consider that the bandwidth benefits are solely a function of storage available to the edge function i.e., at the edge infrastructure. This is reasonable because the bandwidth are not instantaneous in nature but manifest in aggregate use of a web service over time. Even though, compute at the edge infrastructure will be used to service clients and realize those benefits, the main contributor to those benefits is storage. Second, the condition does not paint a complete picture for bandwidth benefits. Even if an edge function may not be able to provide absolute reduction in bandwidth utilization but at the very least, it can provide *schedulability* in the use of bandwidth. It is equally important and valuable to consider this and aggregate benefits of edge functions as it can reduce the use of bandwidth during Internet rush hour (7PM to 11 PM) which occurs at different instances at different locations e.g., the time difference in eastern standard time (EST) and Pacific standard Time (PST).

More generally, as evident from equation (1) and (2), both these benefits are constrained by the limited resources available at the edge infrastructure. More importantly, it also shows that if there are no bandwidth constraints or latency issues then, all the edge function can be provided from remote clouds.

We next discuss the limitations of the model of edge functions next.

Limitations

The presented model captures the benefits of edge functions effectively. Further, it allows us to reason about the generic functionalities that can provide those benefits and reason about technical challenges posed to edge function ecosystem. This model however is limited in the following sense:

- First, the model only considers benefits derived by a single edge function. But by definition the edge is distributed and hence, the edge functions are deployed in a distributed fashion to a number of edge locations. So, aggregating benefits of an edge function from a web service perspective is important. The presented model can be extended to cover that aspect but discussion on that aspect is out of scope of this thesis.

- Second, the presented model does not consider different location of the edge infrastructure i.e., depending on where the edge is located those benefits would differ e.g., if the edge infrastructure is deployed in the cellular towers vs. aggregation points in mobile networks. In a way, the model abstracts this by considering latency to the edge as constant but that can vary depending up on the edge location. The presented model can be extended to cover different scenarios but is out of scope of this thesis.

- Third, although the model provides way to reason about the benefits of edge functions in terms of latency and bandwidth, it alone does not provide a cost model for deployment of edge functions directly. The are a number of factors to consider a cost model for edge functions. First, as discussed above, aggregate benefits of edge functions must be considered. Second, there is an additional web service specific cost function that translates latency and bandwidth benefits to dollars. Finally, since there are no existing deployments of edge infrastructure, there are no model of cost for the use of edge infrastructure. One can use existing web services or CDN cost models (as done in AppSachet work) but they will remain inaccurate in creating a good cost model. Due to these reasons, we kept discussion on cost model for edge functions out of scope this thesis.

Despite these limitations, the model serves as a good starting point for formalization of edge ecosystem. Moreover, it is important to note that the above discussion must not be read as suggesting that latency and bandwidth are the only benefits provided by the use of edge functions. We firmly believe that real potential of edge functions will only be unlocked when there will be edge native applications e.g., the earlier chapters of this thesis demonstrated the benefits of ephemeral app to reduce app discovery barriers or slices delivered from edge functions complementing messaging apps with missing functionalities. Inclusion of application specific logic is what makes edge functions unique and differentiates them from other concepts such as network functions, discussed next.

4.1.3 Edge functions vs. Network functions

It is important to contrast edge functions with other prevalent similar functionality i.e., network functions. We argue that despite similarities, edge functions and network functions serve different purposes. While network functions, typically implemented at Layer 1 to Layer 3, aim to achieve overall better *network* performance or security, edge functions are geared for better *end to end application* performance. The main difference lies in the inclusion of *application specific logic* in edge functions that can abstract the network complexities and optimize application performance.

Lets consider a simple example to clarify this difference: network content caching vs. content replication. While network content caching e.g., using URL as key may be able to provide similar benefits of lower latency to end users and lower utilization of bandwidth for back-end but it can only do it for a non-secure connection where as edge function that acts on behalf of a web service can operate on encrypted traffic and potentially even without violating end to end encryption as described in later part of this thesis. Further, content replication which by definition requires knowledge of application to decide what to replicate and where cannot be implemented as a network function at all. That is exactly where edge functions excel and can provide additional benefits. An example is AppFlux and AppSachet described in earlier chapter that requires knowledge about app usage on end user device to effectively fetch and deliver app updates to the clients on behalf of an app store. This led us to try to identify generic edge functions discussed next.

Table 12: The generic edge functions derived from the model based on their benefits and resource exercised. A_c: Acceleration, B: Buffering, A_g: Aggregation, C: Caching, U: Understudy and S: Streaming.

Benefit provided	**A_c**	**B**	**A_g**	**C**	**U**	**S**
Latency	x	-	x	x	x	x
Bandwidth	-	x	x	x	x	x
Resource exercised						
Compute	x	-	x	-	x	x
Storage	-	x	x	x	x	-

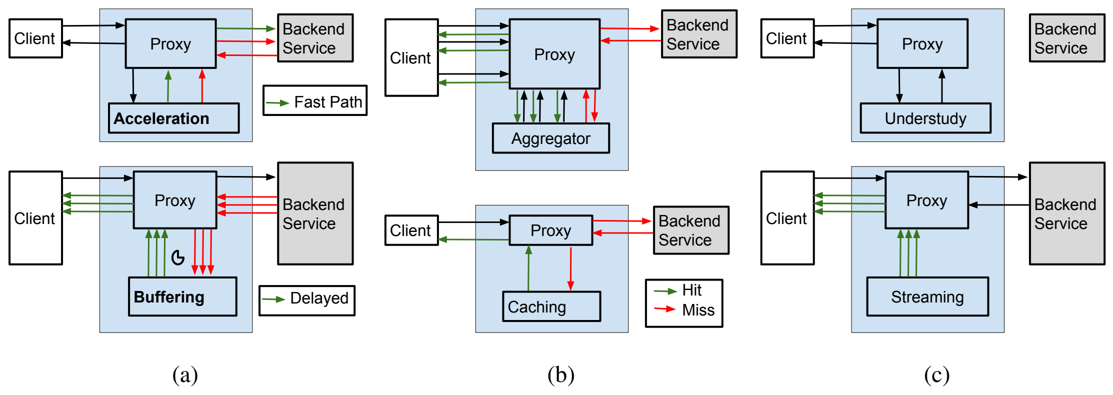


Figure 23: Showing high level design of ABACUS generic edge functions.

4.2 *ABACUS edge functions*

Lack of real world edge infrastructure deployments and hence, edge functions severely limits our ability to reason about them without being affected by idiosyncrasies of particular application being posed as an edge function for a web service. The rationale for defining generic edge functions is also derived from a simple observation is that if there are no latency and/or bandwidth issues, all the functionality implemented as edge function can anyways be provided from the remote clouds. For that reason, we refer to them as ABACUS of edge computing. However, we want to re-iterate that edge functions are not limited to these functionalities.

We observed that different functionalities exercise different edge resource to provide the above mentioned benefits. Table 12 tabulates their benefits and the resource they exercise. In doing this, our rationale was to cover the full spectrum of edge functions characteristics. We discuss them below with an example of real life web service that they can potentially cater to:

- **Acceleration**: An accelerating EF utilizes a *fast path* to inform and/or send relevant information to remote cloud when the client request passes through EF as depicted in 23(i). The fast path can be implemented either by reducing network distance or reducing number of bytes or both e.g., An image analytics service (e.g., Ditto Labs) can detect a product image (coke bottle) in uploaded images from end clients at edge cloud. The send very small packets vs. waiting for the image to be uploaded via UDP using Accelerating EF resulting in faster real time image analytics at backend.

- **Buffering**: A buffering EF stores responses from back end service using EF on behalf of one or more client and delivers the response in a delayed manner as depicted in 23(i). The delay can be based on use connecting to a particular edge node e.g., at home or programmed by back end service to reduce client overheads due to many push notifications. e.g., An end user might authorize a Facebook EF to fetch and buffer his notification and deliver them when he is back home or an app can use buffering EF to reduce push notifications overhead on backend servers for Chrome and native apps.
- **Aggregation**: An aggregating EF stores multiple requests and/or data from clients and sends a single request to backend while removing redundant information from the requests as depicted in 23(ii). The aggregation function can be supplied by backend service or by end user e.g., an edge function as IoT hub could aggregate traffic from different sensors over time while periodically sending information to backend service, resulting in a reduction in bandwidth usage.
- **Caching**: A caching EF stores the responses for client request and then, uses those to service other clients or same client with same requests as depicted in 23(ii). The caching policy can either be service defined e.g., App Sachets [56] which proposes novel caching policies tailored for app installs and updates.
- **Understudy**: An understudy EF provides the same functionality as current backend services but without contacting and/or informing backend services. The functionalities and/or quality of result however can be lower than that provided by remote cloud counter part e.g., hyper local services e.g., YikYak messaging for events or stadiums where everyone is connected to the same edge function.
- **Streaming**: A streaming EF stores bigger content on secure edge node which when authorized by backend services can be streamed to clients. The content streaming logic and/or its format of storage can be defined by backend services in EFs e.g., Ephemeral apps - streaming apps from the edge nodes or simply a distributed video delivery for video streaming.

The seemingly straight forward nature of these generic edge functions can be misleading to assume that there are no technical challenges in realizing them. Contrary to that, (i) these generic edge functions when deployed for some specific web service will carry the sensitive application specific logic which if leaked can lead to major security issues, (ii) there are strict performance requirements for these edge functions either in terms of latency or bandwidth saying which if not achieved absolve the benefits of deploying them in the first place and finally, (iii) these edge functions must be able to ensure end-to-end security guarantees for end users accessing them on behalf of the web service. Learnings from implementing these edge functions and the concrete edge functions for Android app delivery informed our research to develop system level support needed by edge functions presented next.

4.3 Technical challenges

In order to realize an edge function computing ecosystem that can be assimilated in existing computing ecosystem, there needs to be generic and system level support for edge function deployment. We posit the following factors are crucial to develop a general purpose software platform for edge computing.

4.3.1 Low Developer constraints

It is critical that egde function platforms do not put additional constraints on developers in creating new EFs to unlock the real potential of Edge computing. We derive this from from the plethora of tools (Android specific and others) we used to develop edge functions for Android app ecosystem. Without those, we simply would have been bogged down with the implementation intricacies. Simply stated, EF developers must not be require use of specific SDKs, APIs or libraries, etc., outside of the standard system libraries such as libC for Linux and OS-provided system call interfaces. A higher barrier in terms of learning for developers can prove a big hurdle in creating momentum for Edge computing applications. However, we believe that EF development can benefit from using a common set of software patterns corresponding to obvious edge functions such as caching, aggregation, etc.

4.3.2 Fast on-demand provisioning

Wider mobility patterns of end users and resource constraints (storage) on edge obviate the possibility of static provisioning of all possible EFs in all possible locations of edge infrastructure [87]. Further, if the time required to provision an ephemeral app edge function takes more that it is required to install an app, it defeats the whole purpose of having an edge function in the first place. This led us to a requirement that edge function platform must allow fast on-demand provisioning edge functions. An implicit requirement is that developers must be able to package their edge functions in coherent modules. Further, the choice of system software must minimize the time required for EF provisioning. Furthermore, the provisioning must be scalable in terms of concurrent provisioning requests which is a implicit requirement for multi-tenant edge infrastructure.

4.3.3 Privileged software agnostic security

Based on continual security and privacy concern raised by numerous peer reviewers, we concluded that it is crucial for edge functions to address security and privacy aspects. However, we posit that in securing EFs, there are only certain critical functionalities that must be provided – verification of integrity of a provisioned EF, confidentiality of data stored on an edge node. Important to note here is an implicit limit on how much overhead can be tolerated, since too much overhead would defeat the purpose of deploying an EF in the first place. Therefore, securing the complete EF execution via a complete system lock down using encryption are not desired. Another implicit requirement arising out of deployment models of edge infrastructure i.e., whether they are deployed as strategically placed servers or randomly distributed based on individuals or businesses taking franchise of operating an edge node, is that an EF cannot rely on system software to meet its security and privacy goals.

4.3.4 End-to-End semantic preserving communication

Finally, a key challenge, that has remained overlooked in edge computing research, is enabling edge computing is to permit edge functions to operate on encrypted traffic, without breaking the end-to-end security semantics of the applications and the secure protocols they use. Stated simply, edge functions cannot access the end-to-end encrypted traffic without decrypting it. But if they do decrypt it, they inherently trust the privileged software for confidentiality of their secure traffic. The privileged software may be an OS or hypervisor that enables their deployment in multi-tenant settings. This also calls for new forms of security semantics to be defined which explicitly acknowledge presence of edge functions, acting on behalf of web services.

4.4 Chapter summary

In this chapter, we presented a model of edge function to reason about their benefits for existing web services. We presented generic edge functions that we posit can be useful as basis for future research in edge computing by reducing the burden of implementing elaborate use-cases which in our experience sometimes overshadows the discussion on system level issues. Generic edge function allowed us to reason about the system level functionalities needed to develop a generic system software for edge function deployment platform. Based on those functionalities, we outlined the technical challenges associated with developing a generic software platform for edge function deployment and thus, making it possible to translate the edge function benefits to arbitrary web services. However, we want to point out that these challenges are not independent of each other making it important to quantitatively and qualitatively explore the technology space for solutions that address them. We describe such an exploration and the edge function platform developed as part of this thesis in the next chapter.

CHAPTER V

FAST, SCALABLE AND SECURE ONLOADING OF EDGE FUNCTIONS

As discussed in previous chapter, before edge funciton onloading or edge computing in general can deliver on its promises, an *Edge Function Platform or EFP* must address the outined technical challenges. Briefly, for backend web services to utilize edge infrastructure, the EFP must allow them to *dynamically* onload their EFs *quickly*, and the EFPs must be able to *scale* onloading for multiple simultaneous onloading requests. Furthermore, backend services onloading EFs on an edge cloud must be *assured* that their service specific logic included in EFs will not be compromised, that valuable content their EFs store is not stolen, and confidentiality of information about their users is ensured. Since the edge infrastructure is by definition deployed in the wild near end users, backend services cannot trust EFPs or any other privileged software running in the edge cloud for these assurance guarantees. This chapter describes the design of such a software platform.

5.1 Introduction

In this chapter, we describe the design process and design of the software platform – AirBox – that strives to find the sweet spot in meeting the goals laid out for EFP and to provide support for fast, scalable and secure onloading of EFs.

In designing AirBox, we first compared provisioning performance (speed and scalability) for three existing system level mechanisms i.e., virtual machines (cloudlets) vs. OS level containers (docker) vs. user level sand boxes (embassies), listed in Table 13, to conclude that OS containers can provide the right mechanisms and layer for EF provisioning. Further, inspired by recent research [53, 139, 98] which pioneered the use of hardware-level security support such as Intel SGX [119, 26] to run cloud applications securely with an untrusted system software, we propose the use of Intel SGX to provide security and privacy guarantees to EFs running on edge cloud platforms.

Overall, this chapter makes the following contributions:

1. **Design space exploration for speed and scalability**: Using two different hardware platforms with distinct capabilities, we compare the *provisioning performance* in terms of (1) speed and scalability of EF provisioning, (2) invocation speed, and (3) overhead of provisioning (§5.2.2).
2. **Hardware-assisted EF security**. We address EFs' requirements for verifiable integrity, secure execution, and confidentiality for the state stored in the edge clouds, even in the case of physically compromised edge infrastructure, by leveraging upcoming hardware-level security features such as SGX for Intel processors [98]
(§5.2.3).
3. **Design and implementation of an EFP prototype:** We present the design of AirBox—a secure, lightweight, and flexible EFP consisting of two modules: AB console that allows backend service maintainers to deploy and manage their EFs on edge cloud locations, and AB provisioner deployed on edge cloud nodes that allows seamless dynamic provisioning of EFs. In AirBox, we prescribe the anatomy of secure EFs (§5.3), and illustrate their implementation with a benchmark that represents the generic functionalities expected from EFs (§5.4).

Our evaluations show the following benefits:

- AirBox EFs can be provisioned up to 10x faster with only one user when compared to the state of art [87, 93].
- AirBox provisioning scales well in multi-tenant settings with negligible overhead due to its use of SGX.
- We discuss in details how correct usage of SGX leads to EF integrity, their secure execution and data confidentiality with an untrusted EFP.
- We present detailed analysis of the performance overheads caused by use of SGX EF execution, using simple EFs on an OpenSGX [98] platform.

5.2 Design space exploration of technologies for edge function platforms

In our design space exploration, we chose the candidate technologies based on following three considerations:

- *Developer constraints* – which refers to the extra effort required from developers other than for implementing the EF functionality.

- *Provisioning performance* – which refers to the time taken to provision an EF on an edge cloud node when there are multiple simultaneous provisioning requests.
- *Security and Privacy* – which refers to ensuring the integrity of the EF code, confidentiality of the end user interactions, and confidentiality of state saved on an edge cloud platform, without relying on the system software (EFP included) or the physical security of the edge node.

It is important to note that the above list is not a complete list of considerations. There are other factors that can be considered for a comprehensive EFP design, such as dynamic resource management, I/O scheduling [45] and resource isolation [116]. We intentionally omit these from our discussion as they have been extensively studied for data center based systems, and the learning from those studies can be applied for EFs either directly or with some effort.

Based on the chosen considerations, we ruled out a number of technological solutions from our exploration. Specifically, we did not carry out evaluations for JAVA virtual machine-based solutions (e.g., using node.js to implement EFs), application sandboxes that require explicit use of their specific compiler tool chains (e.g., Google's NaCl [153]), and a hypervisor-based unikernel (e.g., Jitsu [114] based on the Xen miniOS kernel) which confines users to a limited system interface and, due to lack of full POSIX support, constraints the libraries which can be used for EF development.

However, we made sure that in carrying out a comprehensive experimental exploration, we chose solutions that operate at different layers of the system software stack. The technology solutions that we identified as potential candidates and which we used in our experimental exploration (listed in Table 13) include: (i) virtual machines (VM) synthesis used in Cloudlets, (ii) OS level containers in form of Docker, and (iii) user level sandbox based on pico-process abstraction emulating a full POSIX interface.

Each of the technologies have trade-offs. For example, the use of virtual machines (VMs) put no constraints on developers but result in large size of VM images to be provisioned. Using application sandboxes avoids OS constraints but puts more constraints on developers, e.g., to use specific tool chains, or a limited system call interface to port existing applications, or results in large binaries due to inclusion of a libOS to be linked. OS containers put constraints on the underlying OS that can be used to develop edge functions. We present our analysis of the chosen considerations next.

Table 13: Enlisting the approaches, we evaluated for their suitability for Edge Function Platform.

Systems Technology	System layer for provisioning	Software stack used in exploration
Virtual Machines	Hypervisor	Cloudlets
Containers	Operating System	Docker
Sandbox	User level application	Embassies

5.2.1 Considering Developer Constraints

In terms of developer constraints, the use of VMs puts no constraints on developers in implementing EFs. Developers can choose the OS, libraries or applications, and package them as VM images which can be delivered and invoked without any issues on hypervisor-based EFPs. The use of OS containers requires developers to implement EFs for a particular OS, e.g., Linux for Docker, but given the increasing penetration of container technology, most popular OSes will support containers [18]. Furthermore, most libraries and/or SDKs are available for all popular OSes, and since EFs typically will resemble backend services in their implementation and/or their dependencies on standard application frameworks (node.js), and software stacks (LAMP stack), this puts little or no constraints on EF developers. The use of application sandboxes either requires the use of specific tool-chains or linking with a platform-level ABI library. For instance, Embassies [94] require pico-process libOS, linking with modified versions of standard libraries such as libC, additional executables for secure execution (e.g., monitor), and a customized elf loader. Despite these requirements, it has been shown that full blown desktop applications can also be run with this approach [95]. It seems that there may be a learning curve, but with appropriate skills, developers can overcome those hurdles.

However, based on our experience during setting up our experiments, we argue that there can be rather subtle assumptions in non-standard solutions. For example, using Cloudlets as described in [87], we realized that the overlay VM created for every client has a static port configuration that needs to be defined as different for all clients even if they will access the same EF. Also, VM overlays that may be created at backend servers are sent to client. The client then has to transfer it to Cloudlet servers, this is a detour we want to avoid. Backend servers can simply send the EF image to Cloudlets server as proposed by onloading approach. Using Embassies took a considerable effort and learning curve in setting it up. The reason is that EFs as Embassies require a new elf format, loader, and effort to port standard libraries to its libOS. In fact, we limited our evaluation to only one

Table 14: Deployment models, capabilities and experimental setup configurations for edge infrastructure.

Type	Deployment scenario	Machine Configuration
Mini-edge	Strategic placed server racks - (Server class machine)	Intel x86-64, 24 CPUs, 1.6 GHz, 50 GB RAM, 4 NUMA nodes, 2 sockets, 6 cores per socket, 2 threads per core, VT-x, L1 (i+d): 64 KB, L2: 256 KB, L3: 12 MB
Micro-edge	Randomly placed standalone servers by businesses or individuals - Desktop class machine.	Intel x86-64, 4 CPUs, 1.6 GHz, 4 GB RAM, VT-x, L1 (i+d): 64 KB, L2: 4096 KB

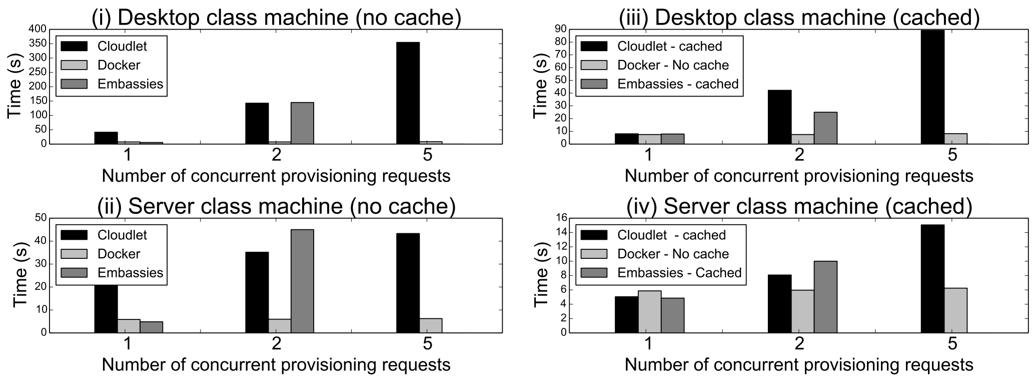
Table 15: Mechanisms employed for security. Note: Invalid for compromised privileged system software.

	Mechanisms		
Requirement	Cloudlet	Docker	Embassies
Provisioning	Full VM image	Image layering	App Image [94, 93]
Invocation	VM boot	Container startup	Run Embassies [71]
Integrity Verification*	Manual	Central registry	cryptographic attestation
Secure Execution*	via Hypervisor	namespaces, cgroups, SELinux, etc.	Not supported
Data confidentiality*	Disk partitions	Disk volumes	encrypted file system [53]
User confidentiality*	mcTLS,split TCP	mcTLS,split TCP	mcTLS,split TCP
Developer Constraint	None	OS	libOS linking, porting

EF or application because we found it very difficult to create and run the same applications across all these solutions. However, to be fair the goal of Embassies was not to support all the applications but to demonstrate the proposed concepts. In contrast, deploying those applications using Docker containers was straightforward. It was as simple as writing a correct docker file to deploy a container containing EF implementation on an edge cloud node.

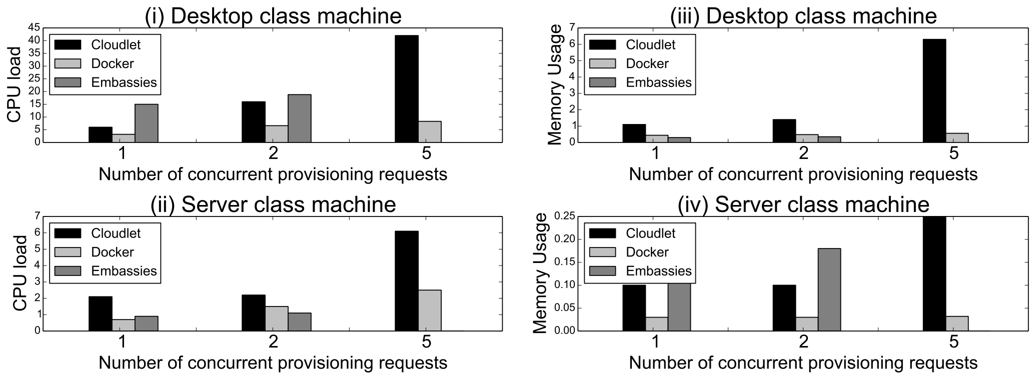
5.2.2 Comparing Provisioning Performance

Concerning provisioning, the use of virtual machines has been proposed to handle just in time dynamic provisioning of ofiňĆoading-based cyber-foraging, to cleanly handle complexities due to their inherent dependence on the mobile devices' operating systems (e.g., Android, iOS, etc.), or the different mechanisms employed in application partitioning and/or ofiňĆoading, which get updated regularly at higher frequency. In contrast, we posit that the EFs will resemble back-end services in their implementation and will exhibit dependencies on standard application frameworks (node.js), software stacks (LAMP stack), etc. Therefore, for provisioning, we consider as viable approaches that use higher layers of the software stack other than OS, such as application sandboxes and OS containers. we believe by choosing a higher layer of software stack may help improve provisioning



(a) (b)

Figure 24: Comparing provisioning time of Docker, Cloudlets, and Embassies without availability of any pre-provisioned base images on (a) desktop and (b) server class machines and with cached base images on (iii) desktop and (iv) server class machines.



(a) (b)

Figure 25: During provisioning of Cloudlets, Docker, and Embassies without availability of base images showing CPU consumption on (a) desktop and (b) server class machines and memory consumption on (iii) desktop and (iv) server class machines.

speed. We present our experimental exploration below.

5.2.2.1 Experimental Setup and Workload:

Table 14 lists the hardware platforms we used in our experiments. Our rationale to choose two different hardware platforms is that each represents a configuration representative of proposed views of edge cloud infrastructure. We used a simple EF application, for which we create instances for all of the chosen technological solutions. We used a simple image inverting application that exposes a command line interface of standard exact image library processing library to perform any image transformation similar to applying filters on an image in Instagram app running on Ubuntu 14.04.

5.2.2.2 Experimental Results:

We present the experimental results as part of our design space exploration below:

Scalable provisioning: Figure 24(a) and Figure 24(b) shows the time taken to fetch or take appropriate actions (e.g., VM synthesis, applying layers on Union FS or saving binaries) to create an EF image that can be booted, with varying the number of simultaneous provisioning requests. In these first experiments, the common parts needed for provisioning, i.e., VM base image for Cloudlets, OS kernel image for containers, and monitor program images for Embassies are not available on the EFP before the start of provisioning.

In addition, the VM base images or the required base software components may be statically pre-provisioned at edge cloud nodes. This puts pressure on the Cloudlet server storage, or additional constraints on developers on what OS/software components they can use to develop EFs, but it can reduce the time taken for provisioning. Figure 24 (iii) and (iv) shows the comparison of provisioning using Docker (no cached images) with Cloudlets using cached base images, and with Embassies using cached base software components. Despite these optimizations, Docker is faster and scales in a much better manner.

Cloudlets suffer from the complexity of the VM synthesis step which requires a VM overlay to be created by a client and then, sent to a Cloudlet server, where binary patching is performed to create a VM image. The overlay file can be significant in size and needs to be sent to the Cloudlet server wasting the limited energy and memory capabilities of the mobile device. Though alternatives Cloudlet models [87] have been proposed where the overlays are distributed from the cloud. Under optimal conditions, with base images cached locally, Cloudlet provisioning times can approach that of Docker (Figure 24 (iii) and (iv)), but the higher overheads of this approach limit scalability, so the Cloudlets approach falls behind when multiple provisioning operations occur concurrently.

Embassies suffer from the sheer size of the resulting executable which we found to be 4 times the size of the same app without running in Embassies. Earlier work showed the delay mostly consists of merkle tree based cryptographic attestation of components of a Embassies app [93]. However, it is important to note that the cryptographic attestation is carried out by privileged software which violates security in our threat model. The figure omits the provisioning measurements for Embassies in case of 5 requests because we were unable to boot 5 simultaneous instances of Embassies in our experiments.

Docker addresses the size issue by using a layered image, made possible by its use of union file system [10], but lacks the security guarantees of applications provisioned in Embassies. Earlier measurements have shown that docker containers can boot faster than VMs [45] also observed in our experiments because VMs need to boot a copy of OS unlike containers which are basically processes. Docker solves the problem of locking of system resources by dynamically reusing the resources for starting multiple tenants using the same application. This also leads to more disk space savings as compared to Cloudlets or Embassies.

Resource utilization in provisioning: We measured the resource consumption during provisioning process in terms of CPU load (Figure 25(a)) and memory usage (Figure 25(b)). From these results, we see that for all hardware capabilities, Docker containers outperform the other solutions for all performance metrics. This provides us with a clear design choice to use containers for AirBox on performance grounds. However, there are security concerns for provisioned Docker containers, as discussed next.

5.2.3 Exploring security options

Concerning security, earlier work provides incomplete solutions to security concerns faced by an EF. Specifically, these solutions leave EFs vulnerable to attacks by malicious privileged software, i.e., OS, EFP, etc. [64], derived from easy physical access to edge cloud infrastructure.

Out of the box, none of the considered solutions have built-in mechanisms to fulfill all security requirements posed by EFs – (i) integrity verification, (ii) execution security and, (iii) data confidentiality without trusting any privileged system software such as hypervisor or host OS.

Important to note here is that EFs *cannot trust EFPs and/or privileged system software* for their integrity verification and/or their stored state due to their proposed deployment models (Table 14). Since the edge cloud is situated in the wild, without any guarantee of physically secure premise like a data center, EFPs can be compromised by malicious parties who gain physical access to edge cloud nodes. This poses severe risks to backend service providers who, by deploying EFs, may exposing their business logic embedded in them. We argue that it is crucial to consider a threat model which covers lago attacks [64] for EFs. This poses additional concerns about the content stored in edge clouds as well as user privacy for users using the edge cloud infrastructure.

There are several available and/or proposed approaches that can be used to address the above mentioned concerns. Specifically,

- For OS level containers, Docker provides a trusted central registry which can be used for integrity verification, leverages kernel features (i.e., namespaces) for isolation among containers, and, can be hardened by enabling SELinux/AppArmour on the host and defining appropriate security policies. By exposing an encrypted file system via VFS, content and/or user privacy can be ensured. However, it is important to note the weaker isolation property of OS containers and the larger attack surface in the form of a shared kernel.
- For full virtualization based systems, integrity verification can be based on checking a hash of a VM image with that provided by a trusted remote registry, implemented as hypervisor based mechanisms. Earlier work like TrustVisor [117] showed that using a formally verified VMM layer, which results in a smaller trusted code base, can be used for securing execution. InkTag [91] proposed the use of para-verification to verify the execution of EFs even with untrusted system layer. However, we argue that verification may not be practical for EFs as a malicious edge cloud node might not implement verification actions as required by Inktag. Since it is not present in a physically secure premise, it may not be practical to force it to use the appropriate host image.
- For application sandboxes such as Embassies [94], isolation semantics similar to data centers can be provided to unmodified desktop applications by running them as web apps on end user machines. Embassies are built on top of the pico-process abstraction [95] that intercept all application interactions with system interface (syscalls). This, combined with integrity verification via cryptographic attestation, can offer strong security semantics. However, Embassies also suffer from the large attack surface in form of shared kernel.

Further, earlier work suggested the use of secure boot [80], verification via redundant execution [143], trust relationships [83], use of Trusted Platform Module (TPM) modules, etc. They provide incomplete solutions for the security concerns faced by EFs, since EFs are deployed in the wild where system software can be compromised or where physical security of the edge cloud cannot be guaranteed. For example, a TPM based approach can be vulnerable to attacks based on physical access, where one could simply take the TPM chip out of platform.

All of the above assume trusted system components i.e., host OS, docker engine, or hypervisor. There are no existing approaches that can provide secure execution without trusting the system. Haven [53] builds on top of the pico-process abstraction and leverages SGX support proposed in Intel's next generation processes to achieve security guarantees for unmodified application without relying on an untrusted host OS. Haven's design suffers from a large attack surface – all system interactions of an application on Haven pass through libOS whose interface exposes a limited syscall interface which is being monitored by the shield module. We posit that covering all possible system interactions may be an overkill for EF performance. Another proposed approach, VC3 [139] reduces the large attack surface and limits performance overhead by partitioning application into trusted and untrusted parts. It proposes to use the SGX support to run only the trusted part and shows a verifiable execution of map-reduce executions on an untrusted cloud. Verification based approaches may be used after the fact but they cannot provide confidentiality of EF state or user requests and certainly leave EF vulnerable to attacks by malicious privileged software, i.e., OS, EFP, etc.

5.2.4 Summary of design space exploration

We derive following conclusions from the above discussion:

- Performant EF provisioning can be realized by using OS level containers. Using containers puts minimal developer constraints and provides fast, scalable provisioning out of the box. However, an additional mechanism is needed for verifying the integrity of the EF being provisioned.
- An EF cannot rely on any trusted components controlled by the system on which it runs. However, it can rely on a processor built-in feature shielded from system software, like Intel's SGX, but it is important to keep the attack surface minimal and to minimize the associate performance overheads.

Based on these observations, we built AirBox on top of Docker while leveraging Intel's SGX to provide security guarantees.

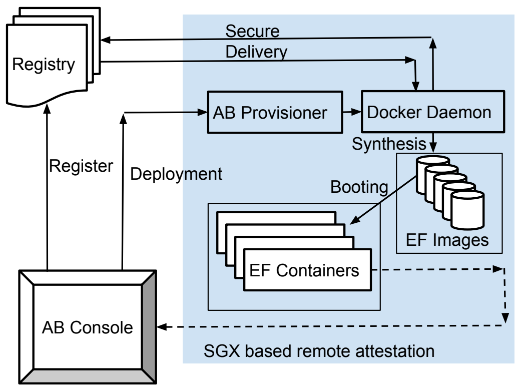
5.2.5 Background on chosen technologies

In this section, we briefly summarize the technologies used in AirBox – the Docker platform along with relevant details about Intel's SGX.

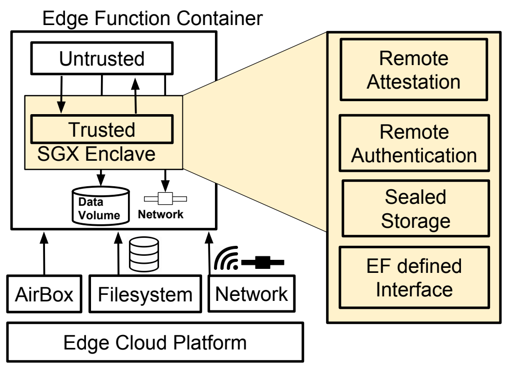
Docker is an open platform for developers and sysadmins to build, ship, and, run distributed applications. Unlike traditional virtualization, containerization takes place at the kernel level. Docker builds on top of these low-level primitives to offer developers a portable format and runtime environment. Docker containers are small, have almost zero memory and CPU overhead, are completely portable and, are designed from the ground up with an application-centric design. Docker leverages the following Linux kernel functionality in its container format: (i) *Kernel namespaces*: Docker uses kernel namespaces to provide a layer of isolation: each aspect of a container runs in its own name space and does not have access outside it; (ii) *Cgroups*: Docker uses the cgroups support in the kernel to allocate hardware resources to containers and, if required, to set up limits and constraints; (iii) *Union File system*: Docker uses kernel support for union file system to create applications layers, making docker containers very lightweight and fast in provisioning. In addition, Docker has built a secure registry service for base container images and other tools simplifying management of distributed applications. Docker provides data volumes and data volume containers to manage data for containers. For additional details, refer to the Docker documentation [17].

Intel SGX is a hardware feature to provide and improve security for applications. SGX offers 4 main features: (i) *Secure ISA extension*: It extends the x86-64 ISA to allow application to instantiate a protected execution environment called an enclave, while only trusting the processor and not system software (hypervisor, OS, frameworks, etc.); (ii) *Remote attestation*: provides a remote attestation feature, in which an enclave can verify the integrity of a target enclave running on another remote SGX-enabled platform; (iii) *Sealing*: allows securely saving enclave data in non-volatile memory for future use, encrypted with a processor-provided sealing key; and (iv) *Memory protection*: When executing in enclave mode, the processor enforces additional checks on memory access using dedicated hardware support, ensuring that only code inside the enclave can access its own enclave region. For details readers are directed to the SGX specification [119, 26].

OpenSGX is a software platform that provides necessary support for SGX application programmers to readily implement and evaluate their applications that leverage trusted execution environment (TEE). OpenSGX [98] supports SGX development by providing: (i) a hardware emulation module, (ii) operating system emulation, (iii) an enclave loader, (iv) a user library, (v) debugging support, and (vi) performance monitoring. We use OpenSGX to prototype and evaluate the performance overhead



(a)



(b)

Figure 26: (a) Showing design of AirBox EF provisioning; (b) Showing overview of secure EF anatomy highlighting the AirBox Open SGX interface.

of providing secure execution in AirBox. Next, we describe the design of AirBox provisioning and anatomy of a secure AirBox EF.

5.3 *AirBox: A platform for Edge Functions*

In AirBox, we focus on the design of two essential elements of EFs, namely *secure provisioning* and *secure EF anatomy*.

5.3.1 Secure Provisioning in AirBox

AirBox provides a centralized backend service, which acts as a central directory of AirBox edge cloud hosts, facilitating their discovery. It also implements AB Console, a web-based management system to let admins manage and deploy services dynamically at remote sites. The actual deployment invokes the Docker mechanisms on each host, after the integrity of the image has been verified using SGX. AirBox can use the Docker container registry service [19], delivery mechanisms that allow docker daemon to pull containers from remote clouds for EF provisioning, and mechanisms implemented in docker engine to handle synthesis of an EF (i.e., download the EF binary and its dependencies.

AB Console can be installed by backend services or can be provided as a cloud based service by a third party that offers edge cloud infrastructure to a number of backend services. Choosing the appropriate AirBox node after discovery is another important concern that needs to be addressed for AirBox. We leave such details for another paper, and focus this paper on the mechanisms of secure provisioning.

A backend service can create its own repository of EF binaries, use already available docker images, or simply register a docker image containing an EF binary and create a docker file describing its dependency in standard ways. There are no additional requirements that AirBox poses on EF developers. To provision an EF, sysadmins can send commands through AB Console to the AB Provisioner modules deployed on edge cloud machines, which in interact with the local Docker daemon. When an EF container is booted on the edge cloud platform, the EF container checks its own integrity using SGX' remote attestation capability.

Since, an EF can tolerate temporarily being unavailable for clients, because in the worst case clients can fall back to remote clouds. We do not consider denial of service attacks in our threat model. Further, delayed verification of execution does not suffice for EFs. Instead, apriori security guarantees are desired for an EF because the edge cloud infrastructure may be deployed in the wild so it can be easily possible to spoof a edge cloud node using replay attacks. Further, edge cloud infrastructure may be deployed in locations with no physical security and it may be possible for attacker to gain access to the hardware (say via connecting a serial port cable). So, it is of utmost importance that that the EF mustn't rely even on AB provisioner for its own integrity check, execution security and data confidentiality.

AirBox EFs can withstand privileged system software based attacks [64] while executing on an untrusted edge cloud infrastructure running untrusted system software (hostOS, EFP). Further, since it is impossible to takeout a subset of instruction set out of a processor, it makes it impossible to compromised EF even if EFP is compromised. Achieving that however is non-trivial and can be achieved by carefully designing EFs and using the AirBox secure interface described next.

5.3.2 AirBox Secure EF Anatomy

There are two main challenges in using SGX to provide data confidentiality for state stored in edge clouds and secure traffic confidentiality to ensure end users privacy that can be violated a result of information leaks from their interaction with EF. First, using SGX to ensure security/privacy is non-trivial in an EF because even if an EF's trusted part is executing within an enclave, it can be compromised by its I/O interface or by privileged system software [64] (e.g., if it relies on system calls that can be logged by the platform). An EF must not leak sensitive information, such as client requests or valuable content that needs to be stored on the edge cloud. Second, running in an SGX enclave leads to overhead due to limited or indirect (via a trampoline) I/O, encryption, etc. Therefore, it is important to minimize the code executing in an enclave for performance reasons.

To address these, EF's in AirBox consist of an untrusted and a trusted part, as shown in Figure 26(ii). The untrusted part handles all the network and storage interactions exposed to an EF by EFP. We assume that all network interaction with clients are over a secure channel established by the TLS protocol. In this manner, we minimize the secure execution of an EF (from within an enclave), thereby reducing the overhead and the attack surface. A secure OPENSGX interface, described next, further addresses the first concern.

5.3.3 AirBox Secure Interface

AirBox uses SGX heavily for security and requires EFs to do so if they need execution security and confidentiality guarantees. SGX support is implemented as instruction set extension and relevant hardware support. The OpenSGX APIs are tied to those hardware capabilities and do not necessarily match the set of intuitive primitives an EF developer may want. AirBox provides a limited but broadly useful and extensible set of higher-level APIs, listed below. These in turn may use multiple native or emulated SGX operations. Thus, AirBox simplifies the use of SGX for EFs and may improve the likelihood that SGX is used correctly to ensure data confidentiality. The interface provides the following capabilities:

- **Remote Attestation:** It allows an EF to verify its integrity using a SGX enabled remote server, using SGX remote attestation feature.

  ```
  airbox_sgx_attest(attest_quote)
  ```

- **Remote Authentication:** It allows an EF to securely query remote SGX enabled remote server for its private key used in a TLS session. It also uses SGX remote attestation feature to fetch the private key securely.

  ```
  airbox_sgx_authenticate(authenticate_quote)
  ```

- **Sealed Storage:** It allows EF to securely read and write on an insecure disk using sealing feature of SGX.

```
airbox_sgx_get(key, key_len)
airbox_sgx_put(key, key_len,*value, *value_len)
airbox_sgx_getkeys(*keys, keys_len)
```

- **EF defined:** It allows an EF to run arbitrary SGX code built using openSGX's libsgx. Examples include implementation of a hash map that can be used to implement a secure customer aggregation logic on an edge cloud.

```
airbox_sgx_run(<module_name>,<conf>)
```

Unique about the AirBox secure interface using hardware functionality is that it enables an EF to keep privileged software blind to the sensitive operations and/or contents.In fact, SGX is not intended to be used for securing I/O operations. AirBox OPENSGX interface allows EF developers to do exactly that but in a secure fashion. Specifically, AirBox uses OPENSGX provided libsgx interface to accomplish the following tasks:

5.3.4 Handling secure traffic

For traffic on a secure channel, the untrusted part of AirBox EF passes to the trusted part the encrypted traffic using the user's key established during TLS negotiations. The trusted part then uses the remote authentication interface to get the current session's private key from a remote SGX enabled backend server, referred to as a *session key*. Once EF has access to session key, it uses it to decrypt incoming requests from clients and responses from backend service. To ensure the session keys persist across a complete session with multiple requests on an untrusted platform, the AirBox EF uses the sealed storage interface to securely store the session key. This rules out any end user privacy concerns that may arise from providing secure traffic access to EFs deployed on EFP, because it disallows any snooping on the traffic even by the privileged system software.

Existing approaches towards allowing edge cloud access to requests over secure channel include the use of (i) homomorphic encryption [141] techniques which have not reached a point where they can be used practically due to their computational intensive nature, (ii) passing key out of bands e.g., mcTLS [123] which would lead to weaker security and privacy guarantees, and (iii) trusting EFP to allow it to access all traffic over secure channel i.e., providing it with a valid root certificate and allowing in to create a split secure connection which can allow a malicious EF to get full access to secure traffic. In addition, the EFs have to trust the EFP when using (ii) and (iii). However, even with AirBox secure interface, threats arising out of statistical analysis of traffic or side channel attacks still persist. We posit that existing differential privacy techniques can be deployed to obfuscate EFs themselves but investigation of this aspect remains part of our future work.

5.3.5 EF State Confidentiality

To perform many useful functions, e.g. caching, an EF will require a secure storage to store the requests and responses on the edge cloud node. Compromising storage can nullify all the effort put in securing traffic. We discuss the how AirBox interface provides secure storage below.

To store data securely on untrusted edge cloud node, an AirBox EF uses the untrusted part to interact with storage exposed to it by the EFP. However, before storing anything on disk, it uses sealed storage interface which gets an enclave specific encryption key, referred to as *sealing key* and encrypts the data it wants to store with it. Then, it passes the encrypted data to the untrusted code and instructs it to write the data to a specific location on insecure disk. To read a file, it instructs the untrusted code to read the encrypted data and pass it in to the SGX enclave, where it is decrypted using the sealing key and can be operated on. This rules out any possibility that content stored on an edge cloud can be stolen even if the privileged software on edge cloud node is compromised.

In designing the above two tasks, there are several non-intuitive design decisions that need to be made. The first concern is around the shared memory allocated to an EF's trusted part or enclave to carry out network and storage I/O via the untrusted host part. In AirBox, we used the trampoline-based approach available in OPENSGX with a maximum buffer size of 5 MB that can be used for I/O for host-enclave communication. If a larger memory is required, then the state or content to be shared has to sliced and re-assembled. This can have performance implications on EF. The second concern is to ensure that host, if compromised, cannot access end user requests or content stored by an AirBox EF. All storage I/O carried out by an EF is designed as a 2 phase process. In the first phase, host communicates meta-data e.g., a path of encrypted blob to be read/written to the storage. The actual I/O e.g., reading/writing the content to storage is carried out in second phase. In second phase, it ii ensured that appropriate encryption/decryption is carried out on the content to ensure functional correctness of an EF by satisfying the following two conditions: (i) content served to end clients is always encrypted with the end user's session key and (ii) anything saved in the storage is always encrypted using sealing key before writing it to disk.

5.4 Implementation

AirBox Console is a web front end implemented similarly to OpenStack Horizon, but simpler and limited in features. AirBox Provisioner is built as an extension of the Docker command line interface to include AirBox specific commands (attestation, authentication) and to accept provisioning requests from the AB Console. In addition, every secure EF image is patched with a binary that loads an SGX enclave to carry out remote attestation on boot up.

The implementation of the AirBox secure interface entails implementing I/O between host and enclave using the OPENSGX-provided I/O trampoline and implementing appropriate encryption/decryption AES routines available in the polarssl library. More importantly, it requires maintaining appropriate ordering of the OPENSGX-provided operations so as ensure security/privacy guarantees. For instance, when an end user request arrives over the network, airbox_sgx_get() uses the following OPENSGX interfaces. First, it uses sgx_enclave_write() and sgx_set_args() to copy data and arguments for AirBox specific commands before entering an enclave. Inside the enclave, it uses sgx_enclave_read() and sgx_get_args() to read the data and commands in enclave memory. Then, it uses 128 bit AES encryption routines provided by the polarssl library. Finally, it returns status and a handle to the value (e.g., stored encrypted file) pointed by that particular key using sgx_set_ret_val() and sgx_enclave_write(). If a match for the key is found during lookup, the host again sends the encrypted value pointed by that handle (e.g., the content of that particular file) and commands using the above mentioned APIs. Inside the enclave, the content of the value is decrypted and re-encrypted appropriately before transferring it to the host using the same APIs. Finally, the response to a user request is sent. Similar steps are performed for the other interfaces. This discussion highlights the fact that using SGX to correctly implement a secure sequence of operations is non-trivial, so the higher-level security primitives provided by AirBox can be of great value to an EF developer.

We implement a suite of generic EFs – the Aggregator, Buffering and Caching EFs to evaluate AirBox. The implementation of ABC edge functions is based on a simple HTTPS proxy that carries

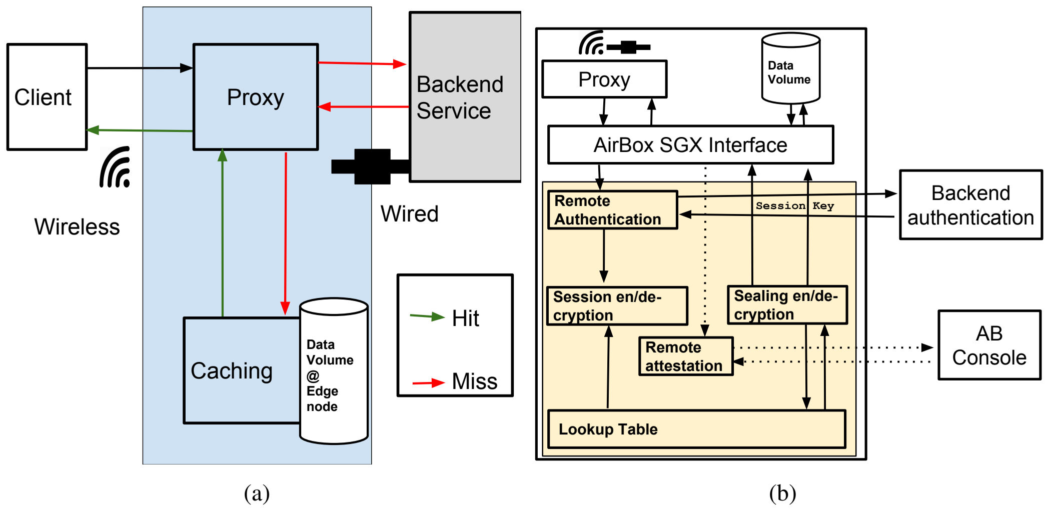


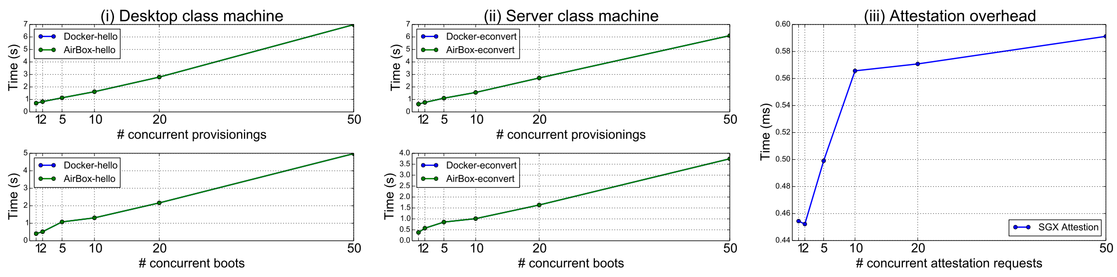
Figure 27: (a) Showing high level design of AirBox EFs; (b) Showing implementation details of a caching EF.

out the following generic operations on web traffic.

We describe the implementation of a secure edge function using the caching EF as an example, as shown in Figure 27. When an end user request is handled at an EF, the untrusted host part implements a proxy functionality setting up the required network connections, and passes the encrypted request to the trusted part running in enclave mode. If a session key for this request is not available, the EF requests it through the remote authentication interface and saves it for the current session from within the enclave. With the key, the EF has access to the request in plain text, e.g., to a URL which is used as the hash table or look up key. The value in that hash table points to a location in the file system partition assigned to the EF instead of to the actual response. In case of a miss, the response for this request is saved on disk at the particular location assigned to the request, but only after encryption using a SGX sealing key is carried out inside the enclave. In case of a hit, the traffic is decrypted, the appropriate encrypted content is read from that path and passed to the trusted enclave code which decrypts it using the sealing key and re-encrypts it using a session key, before responding to the client.

5.5 *Evaluation*

In this section, we present our experimental result using the OpenSGX platform. The evaluation uses as benchmarks the same applications used in evaluating the provisioning performance in §5.2.2



(a) (b) (c)

Figure 28: Comparing provisioning time using Docker and AirBox including SGX attestation for a hello world and an image inverting application vs. increasing number of simultaneous provisioning requests using (a) desktop class machine and (b) server class machine; (c) time spent in attestation command vs. number of simultaneous attestation requests using windows SDK on SGX hardware.

and the simple ABC benchmark EFs, developed to evaluate AirBox. Our future work will consider more elaborate real-world edge functions. Further, since we are using OPENSGX, which is an emulator platform based on QEMU, we believe that it would be unwise to show hard response time measurements or bandwidth saving numbers. We refer to previous work [57, 56, 85] that shows those benefits for any applications based on edge cloud and focus on the effect of using of SGX on provisioning and security. Specifically, we aim to answer the following questions:

1. How much overhead is caused by the SGX attestation during provisioning of AirBox EFs, with varying number of concurrent provisioning requests?
2. How much overhead is caused due to the design of the secure AirBox EF anatomy?
3. How much overhead is added by SGX overhead for generic EFs and how does this vary‘ with workload characteristics?
4. How closely do the SGX overheads observed on OPENSGX resemble the overheads on real hardware?

5.5.1 Provisioning performance

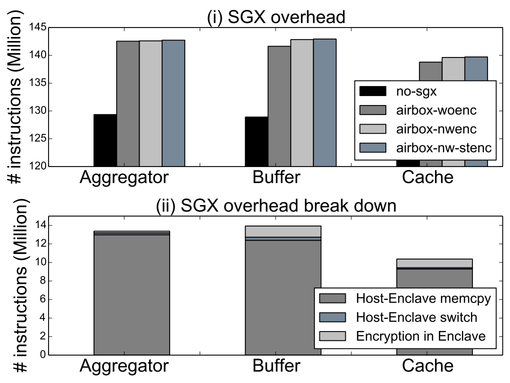
Figure 28(a) and Figure 28(b) show the time spent in provisioning and booting EFs using Docker vs. AirBox, respectively. We carried out the experiments on a server class machine and a desktop class machine with a hello world application, and an image inverting application which uses the command line interface (econvert command) of exact image library processing library. As clear from the figures, the difference in AirBox and Docker provisioning is not visible in any of the cases.

Specifically, the difference is on the order of milliseconds while the attestation requests remains on the order microseconds. Figure 28(c) shows how the time spent in attestation requests varies with number of simultaneous requests.

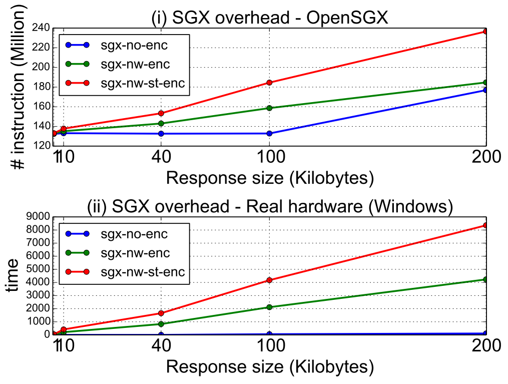
5.5.2 Security and Privacy Overhead

To gauge the overhead associated with ensuring security and end user privacy using Intel SGX, we perform analysis using the ABC EF benchmark. We also implemented an automatic EF load generator that exercises an EF booted on OPENSGX platform. We used a 256 byte request size and 1KB response size to measure the overhead to carry out experiments on a desktop class machine. The size of request and response is important as memory copy and encryption overhead depends on it. We measured the number of CPU cycles consumed during operations of the ABC EFs without using SGX vs. varying level of functionality carried out in an SGX enclave: (i) while handling secure network by exchanging session key from within an enclave to decrypt the end-user request and carry out appropriate EF functions on request and/or response; and (ii) while handling secure network and ensuring that before anything is stored on the file system, it is encrypted inside an enclave using a sealing key. This also entails re-encrypting the stored state with a session key before responding to the client. Figure 29(a) (i) shows the results. To further analyze the overhead, we looked at the break down of where the instructions are spent. As evident from Figure 29(a) (ii), the majority of the overhead arises out of performing host-to-enclave or enclave-to-host memory copies. This is a result of the small size of the request and responses, as opposed to encryption vs. memcpy overhead. This highlights the importance of design choice to facilitate memory copy between host and enclave. To verify it, we measured the number of CPU cycles consumed by an EF while varying the size of the buffer to be transferred between host and enclave. Figure 29(b) shows the CPU cycles spent with varying size of the memory size to be communicated between enclave and host using OPENSGX with different levels of encryption employed by an EF i.e., with only network encrypted and with storage and network both encrypted.

Since OPENSGX is merely an SGX emulator, the measurement may not directly reflect real hardware. To get better sense of the actual overheads, we carried out a similar experiment using the Windows SGX SDK. Note that Linux SGX SDK was not available at the time of the preparation of



(a)



(b)

Figure 29: Showing (a) (i) Airbox overhead in ABC use cases due to the use of SGX for security and privacy using OpenSGX; (ii) Showing the break down of the overhead in ABC use cases and (b) Showing SGX overhead variation with size of data transfer between host and enclave with no encryption, only network interaction encryption and with both network and storage encrypted (i) using OpenSGX and (ii) using Windows SGX SDK.

this paper. Since Windows is not our target platform, we only implement the bench marking cases to measure the performance on real SGX-enabled hardware. The experiment results are shown in Figure 29(b) which shows the time spent in memory copy operations on real hardware where it is difficult to measure CPU cycles spent inside an enclave due to lack of appropriate support/tools[1]. The overhead of including only network encryption and both network encryption and storage encryption are around 0.8 ms and 0.4 ms when the request/response size is up to 200 KB, respectively. The results demonstrate that there is a negligible overhead for performing purely a memory copy operation from host to enclave on the real hardware, implying that majority of the overhead on real hardware would be associated with encryption used inside an SGX enclave. More importantly, evident from the figure is the similar trend with increasing size memory copy with OPENSGX and Windows SGX SDK highlighting that a similar performance can be achieved when deployed using real hardware. However, it is part of our future work to evaluate the performance of real applications such as web caching proxy, firewalls, etc., that utilize the AirBox secure APIs on real SGX hardware.

[1]Instructions such as rdtsc cannot be invoked inside the enclave in SGX revision 1.

5.6 AirBox deployment scenarios

In Mobile networks. We envision AirBox to deliver part of the system level functionalities as described in ETSI's framework and reference architecture for Mobile Edge Computing [21]. In their terminology, the AB Provisioner will be deployed on an edge host providing what is referred to as virtualization support (via Docker containers). AB Console will act as a mobile edge orchestrator either for mobile network operators or third parties, depending on its deployment. Further, AirBox augments the static attestation of edge application (or EF) with SGX based attestation carried out by the EF itself to remove reliance on system level management for its integrity. AirBox with appropriate enhancements to interface with radio access network (RAN) equipment can be deployed in mobile networks.

In Enterprise. Recent trends suggest deployment of on-premise virtual customer premise edge (vCPE) equipment to reduce the number and cost of physical hardware appliances required for hosting value-added features (EFs). AirBox can provide necessary support to quickly, securely provision, and manage those EFs on deployed in enterprise settings. We posit that vCPE providers can integrate AirBox in a straight forward manner enabling them to use containers (with security guarantees) as opposed to current virtual machines to deliver EFs. This can result in a better price-performance ratio for their deployed hardware.

5.7 Chapter Summary

We presented the design space for edge platforms that can execute functionality onloaded on behalf of remote, cloud-based services, in order to address the bandwidth use and address latency requirements of device-cloud interactions. Based on detailed analysis of the current technology space, we observe that OS containers can provide a solution for fast and scalable provisioning of edge functions, with minimal developer constrains. In order to address problems related to the lack of trust among of such edge functions and the underlying platform, we develop a solution that leverages efficient containerization mechanisms (Docker) as well as hardware assisted security (Intel SGX), while also balancing the goals of reduced attack surface and reduced overheads of using trusted execution. The outcome is AirBox – a performant and scalable edge function platform that provides integrity and security guarantees for edge functions.

CHAPTER VI

TOWARDS END-TO-END SECURE EDGE FUNCTIONS

Even with a software platform that allows for efficient and secure deployment of edge functions which is crucial harnessing edge computing infrastructure, there remains a critical, yet common scenario where edge functions are inapplicable: under the encrypted communication. Unfortunately, most web services today are delivered over end-to-end encrypted channels. Worse yet, one report suggests that by the end of 2016, 70% of the global Internet traffic will be encrypted [22], with some networks surpassing 80%.

6.1 Introduction

The key challenge in enabling edge computing is to permit edge functions to operate on encrypted traffic, without breaking the end-to-end security semantics of the applications and the protocols they use, while still providing the expected benefits of reduced latency and reduced backhaul network footprint.

Stated simply, edge functions cannot access the encrypted traffic without decrypting it. But if they do decrypt it, they inherently trust the privileged software for confidentiality of their secure traffic. The privileged software may be an OS or hypervisor that enables their deployment in multi-tenant settings. However, as highlighted by the NSA-linked Cisco middlebox zero day vulnerability [36] incident, the trustworthiness of such privileged software is questionable.

The fundamental problem of establishing trustworthiness of an execution environment controlled by another entity has also manifested itself in other scenarios like cloud computing [50, 120]. However, for edge computing, solving this problem is more challenging because of: (i) *performance constraints:* too much performance overheads to ensure confidentiality of encrypted user traffic can absolve latency benefits expected from the edge functions, and (ii) *the magnitude of security risk:* a compromised edge function can not only compromise that web service but also other edge functions and thus, those web services own by multiple parties.

Performance is of the utmost importance for edge functions because if there are no performance benefits in terms of latency and/or bandwidth savings, then all edge functions can be anyways deployed in centralized remote clouds. Any solution must incur low performance overhead to allow authenticated access to edge functions and should perform well with the concurrent construct.

Regarding security-related compromises, the inadequacy of existing approaches stems from the different operating conditions for edge functions. For instance, it is often assumed that the infrastructure where edge functions are running is trusted and physically secure, e.g., in content delivery networks' data centers where appropriate business agreements could suffice to address security and privacy concerns. This assumption is not true, as by definition, the edge infrastructure implies it is situated at the edge in the wild, e.g., physically insecure aggregation points under lamp posts or publicly accessible cellular towers.

Existing solutions in the networking domain allow middleboxes to securely operate on encrypted traffic which include splitting SSL (or TLS) via based connection termination [109], multiple context encryption [123], and homomorphic encryption [141] schemes. But all of these approaches either incur too much performance overhead for them to be of any practical use or compromise end-to-end security (§6.2.1). Existing solutions in systems space have focused on providing a trusted execution environment (TEE) by reducing the trusted entities to a small verifiable code base [51, 65, 90, 146, 152, 155, 132] and trusting only the hardware [52, 126, 138]; they lack discussion on how to enable establish trustworthiness of communication channels with such primitives, which is an inseparable and critical part of edge computation. More importantly, even for systems that provide the capability to allow backend web services to measure the trustworthiness of the edge functions through remote attestation [149, 89]; without careful design, communication to the TEE may be subject to two types of attacks: time-to-check-to-time-to-use (TOCTTOU) and cuckoo attack [128] (§6.2.1).

6.2 SPX: Preserving End-to-End Security Semantics in Edge Functions

To address these challenges, we present the Secure Protocol Extensions (SPX) for existing secure protocols to allow edge functions to operate on encrypted traffic in collaboration with backend services. SPX uses hardware-assisted TEE such as Intel SGX in the edge infrastructure, where edge functions are run; backend infrastructure, where web services are run. They collaboratively allow edge functions to access encrypted traffic while preserving what we refer to as End-to-Edge-to-End (E^3) secure semantics (§6.2.2). Finally, the SPX not only should it allow a web service to measure the edge functions, but it allows edge functions to check the integrity of web services.

Achieving this however, does not mandate any changes to end user devices. We prototyped SPX for TLS (§6.3.3) and protocols based on the Noise framework (§6.3.4). We demonstrate their efficiency in terms of the client's setup connection time, and the overhead, which is observed by the client and server on the opposite ends of the edge (§6.5.2). Furthermore, to highlight its efficiency and scalability, we evaluate the them with realistic workloads. (§6.5.1).

In summary, this chapter describes the following research contributions:

- A design framework for creating secure protocol extension SPX for existing protocols, while still preserving E^3 security semantics.
- We demonstrate its use by prototyping TLX (SPX enabled TLS) and NoiXe (SPX enabled noise-based protocol framework). We then demonstrate that deploying SPX does not require any visible changes on the end user's side.
- We outline the design choices in constructing SPX to address protocol performance, which ensures secure semantics and the limitation of trusted execution environments (i.e., Intel SGX).
- With our experimental evaluation, we show that SPX acquires an overhead of up to 20% in handshake performance. Lastly, we demonstrate that this overhead is amortized in bigger size file transfers and web page loading.

6.2.1 Inadequacy of existing solutions

Today, end-to-end encryption is prevalent for ensuring the confidentiality of communication. Consumer web services employ Transport Layer Security (TLS) to guarantee confidentiality in communication to their users; Whatsapp uses noise pipes for its encrypted chat's transport security [47], and Cisco proposes an IoT security framework [78]. We posit that application defined security protocols will become more pervasive for emerging applications such as those proposed for the Internet of Things, driver-less cars, etc. However, to grant edge functions access to encrypted traffic using existing solutions have a undesirable side effect of compromising security or compromising performance.

Network solutions. Existing solutions in the networking domain either incur too much performance overhead for them to be of any practical use or compromise end-to-end security.

Split TLS connection: It means that clients' secure connection terminates at the edge location [109]. Edge functions are pre-provisioned with a valid authentication token, e.g., certificate, to act on behalf of a backend service. This means that data transfers can be made available in plaintext where the connection is split. This approach is employed conventionally by content delivery networks (CDN) which deliver content on behalf of their clients and by mobile network operators in their core network to optimize the delivery of content (e.g., down scaling video for mobile device).

Multiple context encryption: Recent work [123] proposed the use of multi-context encryption scheme as extension of TLS protocol. It involves encrypting the data, or part of the data, with different keys (or contexts). Then, selectively sharing those keys with middleboxes to allow them access to pre-defined part of the traffic. With this, the edge function can only access the encrypted traffic partially assuming that there are multiple contexts. If there is only one context then otherwise it is the same thing as sharing the key out of band with middleboxes and hence, compromising end to end security. This approach has several limitations. First, using it requires updating the end user's devices, middleboxes, and servers. Secondly, it requires applications to appropriately partition and tag the data sent to the multiple contexts (or encrypt them with different keys). The partitioning task is non-trivial. Third, because it shares keys to partial content, it makes the traffic more prone to side channel attacks and limits the functionality due to partial access. Lastly, if full access is granted, then it violates the end-to-end security.

Homomorphic encryption: Another recently proposed approach [142] suggests the use of homomorphic encryption to allow introspection in the middleboxes. However, homomorphic based encryption computation is far from practical in terms of its computational requirements to be a viable option for edge computing applications.

Furthermore, we argue that assuming a trusted edge infrastructure, including trusted privileged software (OS, hypervisor) that hosts edge functions, is not desirable in edge computing scenarios. This renders edge functions deployed in multi-tenant setting prone to a malicious edge function which can gain root privileges via a vulenerabilty, i.e., includes attacks by a compromised OS [64]. For instance, a malicious edge function can act as a Netflix edge function to its Netflix web service;

thereby request and steal all the videos from the Netflix web service.

TEE-based solution. It may appear that the use of a trusted execution environments with remote attestation feature would solve the problem. Unfortunately, it will not. The usual approach to authentication, e.g., using a shared secret or public key infrastructure, does not suffice for edge computing because it requires an existing trusted channel to deliver the shared secret or the private of a certificate. But no such channel exists if the edge infrastructure provider or malicious edge function with root privileges (MEF), who controls both the edge infrastructure and the network, is not trusted. Consequently, the only reliable way for authentication, is by using a unforgeable report in the form of remote attestation. This is typically created by a correct or benign edge function (BEF) using trusted execution environment. Based on this unforgeable evidence, one can be reassured whether an edge function can be trusted or not.

However, if a communication channel is not carefully binded to the authentication, two general types of attacks become possible:

TOCTTOU attack: A communication channel is vulnerable to a TOCTTOU attack if the attestation is performed *before* establishing the communication channel. Consider the following: A web service initializes a remote attestation request to measure the target edge function. Next, the MEF routes this request to the correct BEF edge function. The BEF responds with an expected measurement and the web service initializes another encrypted communication channel to edge function. To provide confidentiality and integrity (e.g., using the TLS protocol), MEF routes the new communication to itself instead. This attack is possible for two reasons. First, the MEF has full control of its internal network, so it can route network packages to whichever edge function it wants. Secondly, because there exists a lack of a trusted channel to deliver the certificate, it is impossible to rely on TLS to ensure both MEF are BEF are the same.

Cuckoo attack: A communication channel is vulnerable to cuckoo attack if the attestation is performed *after* establishing the communication channel. When a web service initializes an encrypted channel to an edge function, the MEF routes the communication to itself. The web service initializes a remote attestation request over the established channel. MEF forwards the attestation request to a BEF, and the BEF responds with a correct measurement to MEF, which forwards the correct measurement to the web service. The cause of this attack is similar to the TOCTTOU attack: MEF can arbitrarily route network packages and TLS cannot distinguish between the MEF and the BEF.

Using TEE has inherent performance overhead due to memory encryption, limited or no I/O access in trusted mode, limited addressable memory (e.g., Intel's 6th generation processors have a maximum of 128 MB addressable by an enclave), limited number of concurrent trusted execution (e.g., Intel's 6th generation processors can support a maximum of 12 enclaves), and compared to canonical threading model (e.g pthread) multi-threading within an enclave is not trivial.

6.2.2 E^3 semantics for secure protocol extensions

E^3 properties. Before introducing our assumptions and design goals, it is important to first define the E^3 security properties discussed further in §6.3. We consider an SPX to be satisfying E^3 properties if the following two conditions hold:

1. The SPX preserves end-to-end security properties for encrypted network traffic transmitted on it i.e., prevents access to encrypted data by any unauthorized party; and
2. The SPX does not mandate trusting the privileged entities (e.g., system software) in enabling access to encrypted traffic to authorized entities.

For clarification, consider a simple example of Netflix which deploys its edge function that caches videos on an untrusted edge infrastructure and uses the TLS protocol to secure the connection to users. When a client connects to a Netflix server, it is desired that the content is served from the edge function for obvious latency and bandwidth benefits. However, Netflix will only want to stream videos to the clients authorized by its backend service. In this example, a connection must be setup between the client and backend server while the edge function serves the content. This is different from the scenario of the content delivery networks in which the secure connections of the clients terminate at CDN nodes. It should be noted that authorization of edge to act on behalf of a backend server is an automatic side effect if a protocol extension provides E^3 guarantees.

6.2.3 Assumptions, Threat Model and Goals

Assumptions. We make the following three assumptions. First, we assume that backend services are willing to deploy edge functions on open multi-tenant edge infrastructures. Edge functions handle end user requests over secure protocols to provide reduced latency to their users and bandwidth benefits to them. At the same time, they want E^3 guarantees to preserve the privacy of their users and to secure their states that lie within edge functions. Second, we assume that when a client try to access a web service, the connection is transparently directed to an edge function. The redirection can be a DNS-based redirection, or an explicit mechanism deployed by the edge infrastructure e.g., redirecting RAN packets to appropriate edge functions by some sorts of match-forward rule, or HTTPS based redirection (although this may absolve the benefit of deploying edge functions due to latency concerns). The mechanism used for redirection is orthogonal to the design of SPX and we assume that somehow the packets are redirected to a SPX enabled edge function. Third, we assume that security guarantees provided by existing protocols in end-to-end client server communication hold, e.g., TLS or noise based protocols provide the required end-to-end security. SPX relies on that to create extensions. Finally, we assume availability of an OS-agnostic secure channel between the edge and the backend servers e.g., Intel SGX remote attestation feature. With these assumptions, we next specify explicitly our threat model for SPX.

Threat Model. Different from traditional network threat model where an adversary usually is in the middle and launches attacks over the intermittent traffic, we consider a stronger threat model where an adversary has a full control over the edge infrastructure including the operating system, except for the isolated execution environment (TEE) which runs a trusted program. The adversary can freely monitor, intercept, and forward the data over the communication channel between the OS and TEE. For example, if an edge function running in a TEE tries to open a channel to OS in order to talk to remote clients, the adversary who controls the OS also has the full control over that channel. However, the adversary cannot mount any attack directly on the TEE, since we assume TEE is under a strong protection against all external software accesses. We do not consider denial of service attacks of edge functions that can be mounted by the operating system.

Design Goals. Giving the above assumptions, this work aims to achieve the following goals. The first goal of this work is to articulate a generic protocol extension to an existing secure protocol that can be applied to an edge computing network while providing *End-Edge-End or E^3 secure properties*. The second goal is ensuring *interoperability* of SPX with existing protocols, i.e., no changes should be required in protocol stacks deployed on end user devices. The third goal is minimizing the *deployment* barrier for SPX in the existing eco-system, i.e., integrating and deploying

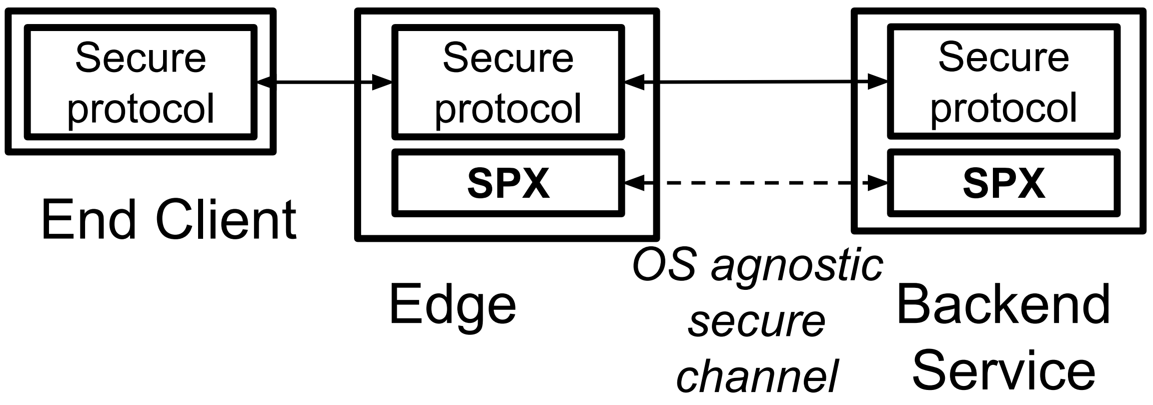


Figure 30: Showing the overview of SPX approach.

those extensions should take reasonable amount of effort in edge functions and backend services. The final goal concerns the *performance* impact of using SPX to provide E^3 guarantees, i.e., the created protocol extensions must put minimal performance overheads on end clients, on edge functions, and servers.

6.3 Designing SPX with E^3 semantics

Figure 30 shows the simplified view of how SPX can be realized for a general secure protocol, using OS-agnostic communication channel secured by remote attestation provided by the TEEs. We next discuss the edge- and server-side details of SPX during different phases of a protocol lifetime.

6.3.1 A Generic protocol

What SPX accomplishes is to allow the communication channel between the client and the edge function to be secured transparently as if the client was directly communicating with the server. This is achieved in two steps. First, SPX utilizes TEEs to perform the encryption so an MEF cannot access the encrypted network traffic. Second, SPX utilizes an attestation-binded handshake protocol to transfer the session key of the supported protocol from the server to the edge. As a result, when establishing the secure channel, the client still authenticates using the server's credentials, so the edge function is transparent, e.g., during a TLS handshake, the client will see and authenticate the edge function using the server's certificate instead of the edge provider's certificate. Next, the authentication key of SPX-enabled secure protocols (e.g., the private key of the server's certificate) never leaves the server so the risk of identity theft is low. Finally, by binding the key exchange between the edge function and the server with an attestation, SPX is resistant to the two kinds of attacks discussed earlier in §6.2.1. Next, we describe in detail how the SPX's handshake protocol works.

Detect: A SPX enabled edge function must be able to detect the client's session initiation message which is usually transferred in plain text.

Relay: Once the edge function detects an initialization of SPX-enabled protocol, it relays the initialization message to the server and behaves as a transparent proxy before the session is established. It is important that edge function monitors the whole handshake process so as to extract relevant protocol state (e.g., encryption suite) that is necessary to talk to the client.

Bind: Piggybacking with the relayed traffic, the edge function sends additional message that binds the attestation with a ephemeral public key to the server. Note that, the ephemeral key pair are generated inside the enclave and will be deleted after the handshake completed. This is the step where the two aforementioned attacks are prevented.

Forward: To maintain transparency, the edge function must not communicate any SPX-related information and/or clears any marks in the protocol messages to be handled at the edge before forwarding them to clients. This function is important to avoid interoperability and deployment concerns with SPX.

Grant: When the server finishes the session establishment and the authentication of the edge function is successful (by verifying the attestation report), the server transfers the session key, which is encrypted by the ephemeral public key contained in the attestation, to the edge function through the key exchange channel. On receiving the session key message, an edge function decrypts the key using corresponding ephemeral private key, registers the session, and begins serving the request of clients thereafter.

After successful creation of a secure session, edge function maintains the state of the session unless explicitly directed to discard. SPX also maintains the state per connection and supports resume operations supported by the protocol. On carrying out these function in appropriate manner, the connection between the client and edge function would satisfy the E^3 security properties.

6.3.2 Design Choices

The following design choices are carefully considered while designing SPX:

Channel binding: The key to prevent the TOCTTOU and cuckoo attacks is to bind an attestation with the corresponding communication channel. In SPX, this is done by including the ephemeral public key in the attestation so when the server verifies the integrity of the edge function it can check (1) whether the edge function is indeed running inside a TEE, (2) the integrity of the edge function, and (3) the ephemeral key pair is indeed generated inside the TEE so no other entity has access to it.

Channel relaying: After the detection, an edge function can carry on a handshake with server without involving clients to carry out binding. After the binding is successful, it acts as a transparent forwarder for client-server handshake messages. At appropriate instances, server shares the session state and session key but before communicating the secure connection established to client. This allows SPX enabled edge function to relay the same communication channel for its own and clients handshake. This is important because requiring additional connections is undesirable from edge function and server resource use perspective.

Piggybacking. SPX messages are piggybacked with existing protocol messages as much as possible as opposed to detached messages. SPX piggybacked markers trigger an SPX function at edge functions or server. This is important as it ensures that SPX protocols do not make existing protocols vulnerable to man-in-the-middle attacks and to keep lower overhead on edge and server side while at the same time keep client side engaged to avoid timeouts.

Protocol replication. Wherever possible, SPX replicates the protocol's abstract state at the edge function instead of having the edge function to maintain all state variables, replicate the crypto functions. SPX enabled edge functions only maintain protocol state and request the session key at appropriate state of the protocol via remote attestation. This allows SPX to keep the computational overhead associated with carrying out those functions at the edge.

Limitations of TEE hardware. It is important to consider limitation of TEE hardware e.g., (i) 6th generation Intel processors have only 128 MB of memory addressable to all the enclaves running, (ii) the number of enclave instantiations running simultaneously is limited, and (iii) running an SGX thread consumes one of the logical CPUs and make it unavailable to other threads. Although these are soft limitations that may be addressed in future generation of processors but a practical SPX should be able to overcome these limitations. To address (i) SPX uses a spilling mechanism i.e., session look up table spills over to host memory if the memory available to enclave is not available. However, to ensure E^3 guarantees, SPX uses another TEE feature sealing to store session information. To guarantee the confidentiality and integrity of the sealed information, sealing encrypts the information using the hardware generated sealing key. By doing this, with a small performance penalty, SPX-enabled edge function is only limited by the amount of *host* memory as opposed to memory addressable by a TEE. To mitigate impact of (ii) SPX enabled edge function requires only one enclave instantiation per edge function as opposed to one enclave instantiation per session. However, SPX cannot share enclaves with two different edge functions without compromising E^3 guarantees because in that case two edge functions can generate the same ephemeral key and same attestation reports. To mitigate impact of (iii), we need to make sure that the time required in binding is kept to minimum so that the the edge function waiting for attestation response including session keys finishes as quickly as possible accomplished by monitoring the state of the handshake.

6.3.3 TLX = TLS + SPX

TLS/SSL [148] is a widely used secure protocol that provides end-to-end confidentiality and integrity guarantee for network traffics. The protocol has two types subprotocols: the handshake protocols and the record protocol. The handshake protocols is for negotiating security parameters for the record protocol, authenticating two peers, reporting errors, etc. It is also the key step to prevent MITM attacks. In particular, by using public key encryption, TLS guarantees that only the two communication peers have access to the randomly generated session key that is used in the record protocol. In TLX, we utilize SPX to securely transfer this secret key from the server to the edge function without trusting the edge provider. Figure 31 shows the TLX protocol as described below:

1. Detect function is carried out by an edge function detects a `ClientHello`, SPX adds a request as a TLS extension in the `ClientHello` message and forward the message to the server.
2. Having received the request, the server response by including a response in its `ServerHello` message, again, as a TLS extension. The response can be `OK` or `Not Capable`. In case of `OK`, a challenge (nonce) is sent to guarantee the freshness of the attestation and prevent replay attacks

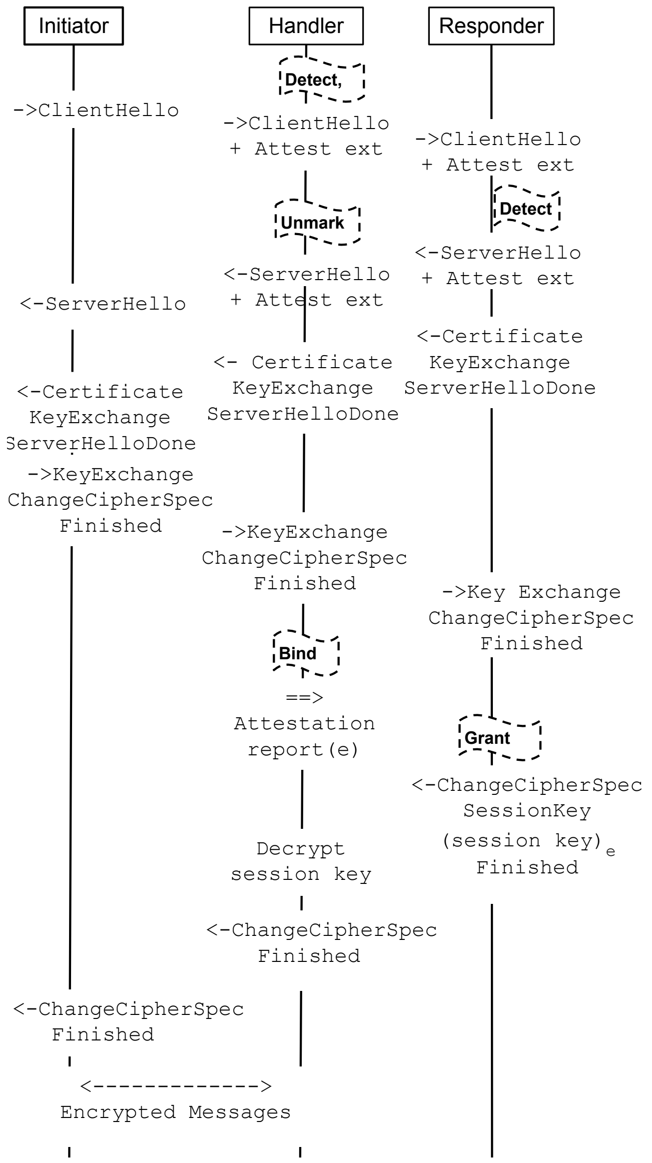


Figure 31: Showing the SPX enabled TLS protocol: TLX handshake protocol.

and a signed ephemeral public key (similar to the `ServerKeyExchange` message) to establish the key exchange channel.

3. Having received the response from the server, SPX strips the server response from the `ServerHello` message and relays the rest messages to the client.

4. SPX validates the certificate, checks whether it is from the trusted server (similar to public key pinning [75]), and verifies the ephemeral key is signed by the public key corresponds to the certificate. This is important to prevent MITM attacks between the edge function and the server [111].

5. While waiting for the client to response, SPX generates a fresh pair of ephemeral keys and generate an attestation that includes both the public key and the challenge (nonce). Then it sends the attestation and the public key to the server together with the client response.
6. Having received the attestation, the server verifies its freshness, correctness as well as the legitimacy of the public key. Once every check clears, the server then send the session key to the edge function encrypted with the exchanged ephemeral key, as well as `ChangeCipherSpec` and `Finished` messages. Please note that for transparency, all SPX-related messages will not be included when calculating the hash of handshake messages.
7. SPX strips and decrypts the session key and forward the rest messages to the client. From now on, the edge function can securely communicate with the client using the TLS record protocol with agreed cipher spec.

The handshake will be aborted if either the normal TLS handshake fails or an attestation-related error occurs.

6.3.4 NoiXe = Noise + SPX

Noise is a framework for crypto protocols based on Diffie-Hellman key agreement. Noise can describe protocols that consist of a single message as well as interactive protocols. A Noise handshake is described by a simple language consisting of tokens arranged into message patterns. Message patterns are arranged into handshake patterns. A handshake pattern specifies the sequential exchange of messages exchanged between a initiator (client) and responder (server) that comprise a handshake. A handshake pattern can be instantiated by crypto functions (DH functions, cipher functions, and a hash function) to give a concrete Noise protocol. A noise protocol starts with prologue data and some patterns require pre-messages to be exchanged between the parties. In a nutshell, message include `e:` ephemeral key, `s:` static key and `dhee, dhse, dhes, dhss:` sender performs a DH between its corresponding local, remote and performs hash). Readers are directed to the Noise specification for further details about the framework and language to describe protocols [130].

Applications can choose to specify and use their own defined protocol between their end users and the web service. NoiXe allows edge functions to support any secure protocol that can be built using noise protocol framework. A noise protocol starts with prologue exchanged between a server and client. For example, suppose a server communicates a list of Noise protocols that it supports to clients. Application on client side will then choose among them and execute that protocol to communicate with server. SPX uses prologue to perform detect function on edge and server side. When a client tries to connect to a server using a noise protocol, it is directed to a NoiXe enabled edge function, it leverages the hand shake pattern information of the protocol being used to choose appropriate messages to carry out its other functions. First, it creates a secure channel with server using the same protocol as requested by the client. The, edge function and server carry out the bind function i.e., uses the channel to send and receive attestation report created in a secure enclave. On successful mutual attestation, a NoiXe enabled edge function relays the same channel to client and while acting as a noise proxy for it to carry out its handshake with the server. This ensures that security expectations of the clients are honoured and the channel which was binded is the one that is getting used for client server communication.

On edge function side, after detect function, the protocol NoiXe capable edge function initializes a handshake state object for that protocol and keeps it up to date with the state on the server side while acting as a forwarding proxy for communication between web service and client. Since, noise uses symmetrical encryption, there is no need to replicate the execution of protocol on edge function side instead, it selects the appropriate messages from the protocol's handshake message pattern (part of Noise handshake state) to get required information from server e.g., current hash state. Finally, the session keys are granted by server to edge function before the last message is exchanged between server and client as part of another attestation report to prevent eves dropping or impersonation. In cases, the last message in the handshake pattern is from the client side, server grants the session keys after it finishes the crypto function are finished. The handshake is only carried out whenever an attestation request can be satisfied (indicated by attestation extension). The handshake is aborted if a requested attestation is not received or is invalid, or the noise session information does not match the signature in the attestation.

Figure 32 shows the handshake patterns Noise_XX between a initiator (client) and responder (server) along with the SPX functions that are carried out between handler (edge) and responder to achieve SPX goals. Noise_XX is used to start a compound protocol called noise pipe. Noise pipes do not assume any previous communication between client and server. The Noise_XX handshake

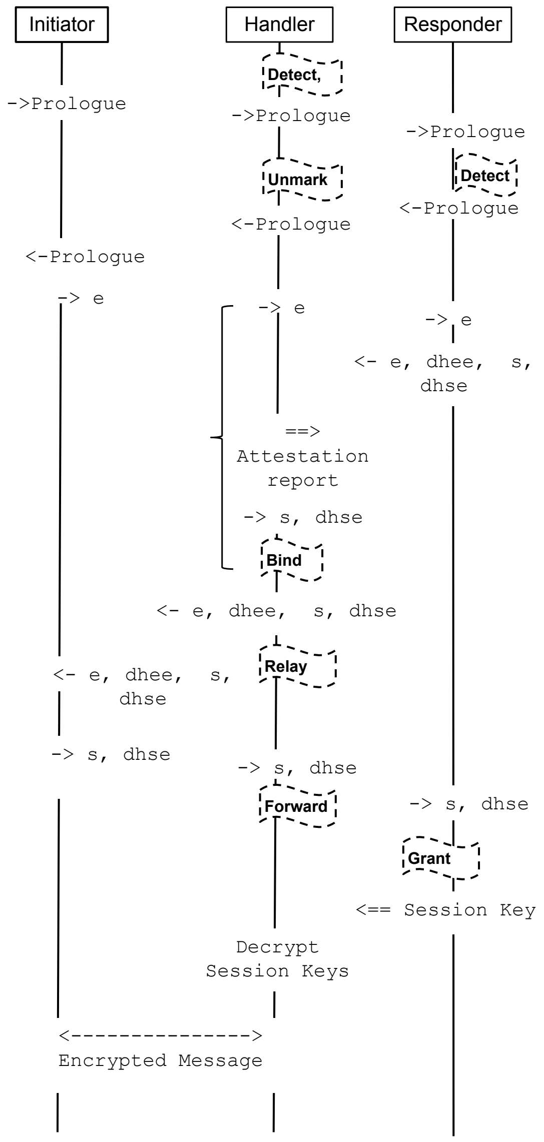


Figure 32: Showing the Noise_XX pattern handshake with SPX enabled Noise framework.

pattern supports mutual authentication and transmission of static public keys which are stored and used as pre-messages in the next Noise_IK pattern to create the session. Important to note is that bind happens only after fir DH function is performed. Identifying this step requires knowledge about the protocol and NoiXe preserves it by piggybacking the attestation with DHed message. Other protocols are supported in similar manner.

6.4 Implementation

We implemented SPX prototypes for Ubuntu 14.04 using Intel SGX linux SDK. We highlight interesting implementation details for TLS and Noise protocols below:

TLX. We prototyped TLX by modifying TLS implementation on top of mbedTLS libraries. We leveraged the extension feature introduced in TLS 1.2 protocol to integrate attestation request and verification into handshake. We then ported the modified library into an enclave to support SPX functionality.

NoiXe. We modified the Noise protocol framework to support a new role of 'handler' to create an intelligent Noise edge function that allows replication of the state of noise protocols. To run noise protocols in an SGX enclave, we ported the full Noise protocol framework to run inside an enclave including all its crypto functions. Using it, we implemented a proxy that runs as the edge function and a server that uses the SPX enabled Noise framework - NoiXe. We had to disable use of vectorization in blake2 and ChaChaPoly because SSE2 optimizations are not supported in an SGX enclave with 6th generation Intel processors.

Remote Attestation for Enclave. Remote attestation in Intel SGX is designed to use trusted servers i.e., Intel servers or other parties that are granted the processor secrete keys under suitable agreements, to verify the attestation measurements. However, in our implementation, we bypassed the use of those servers to simplify our implementation. But it doesn't lead to any loss of generality of our prototype or evaluations because we posit that parties deploying edge functions would act as those trusted entities as per Intel.

6.5 Evaluation

A SPX enabled protocol preserves E^3 semantics while allowing edge function full access to the web traffic. Careful design of additional messages exchanged between edge function and backend servers and secure execution guarantees provided by Intel SGX make it possible and transparent to end users. In this section, we evaluate the overhead incurred to carry out those steps.

Experimental Setup. Table 16 lists the experimental setup used in the experiments. We used machines with 6th generation Intel processors that have SGX support to prototype edge function and remote server. We implemented a TCP client, proxy (edge function) and server which communicate

Table 16: Experimental setup for SPX evaluation.

Setup	Configuration
Client	Intel(R) Xeon(R) CPU E5-1620 v2 @ 3.70GHz, 16GB DIMM DDR3 Synchronous RAM, Intel Ethernet Connection (2) I219-V
Edge	Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz, 16GB DIMM DDR3 Synchronous RAM, 128 MB configured for SGX, Intel Ethernet Connection (2) I219-V
Server	Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz, 16GB DIMM DDR3 Synchronous RAM, 128 MB configured for SGX, Intel Ethernet Connection (2) I219-V
Network	LAN over 512 Mbps connection. RTT (Client-Proxy: 0.964) (Proxy-Server 0.901) (Client-Server 0.962)

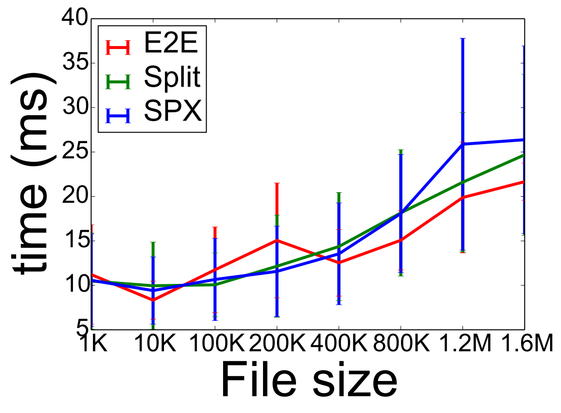
Table 17: Showing handshake patterns of Noise interactive protocols used in experiments along with reference to understand handshake pattern from protocol name.

Noise protocol	Handshake pattern	Reference
Noise_IX	(-> e, s); (<- e, dhee, dhes, s, dhse)	First letter for Initiator
Noise_NK	(<- s); (...) | (-> e, dhes); (<- e, dhee)	Second letter for Responder
Noise_XN	(-> e); (<- e, dhee); (-> s, dhse)	N = No static key
Noise_NN	(-> e); (<- e, dhee)	K = Static key Known
Noise_NX	(-> e); (<- e, dhee, s, dhse)	X = Static key Xmitted ("transmitted")
Noise_IN	(-> e, s); (<- e, dhee, dhes)	I = Static key Immediately transmitted

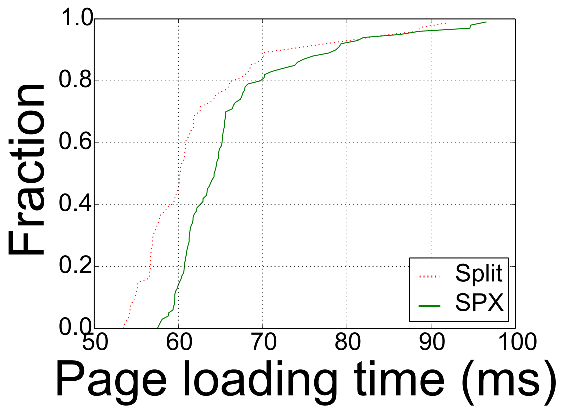
using TLS (mbedTLS library) and Noise protocols using standard framework as our baselines. Then, we added SPX capability to edge and server side i.e., implemented the portion that handles encryption state (proxy and server both) to run in an SGX enclave used TLX (modified mbedTLS library) and NoiXe (modified Noise framework libraries). We used 128 bit AES encryption in RSA for TLS with a certificate of size 3KB and ChaChaPoly cipher, 256 SHA2, 56 bit curve488 key for Noise based protocols. In both cases, we used a 512 byte attestation report generated by processor inside an SGX enclave. We used echo protocol over TCP socket connection to carry out our experiments.

6.5.1 End-to-End Evaluation

File Transfer. Figure 33(a) and Figure 34 compare the time taken to transfer files of different sizes with baseline. We chose the file sizes for the experiment from the average size of objects transferred in top 100, top 1000 and all websites from HTTP Archive [43]. The experimental results shown are averaged over 20 runs along with one standard deviation. As evident from the figure that SPX incurs modest overhead of 12-15% over split proxy baseline. This can be attributed to execution inside a SGX enclave. Partly, inherent SGX overhead and partly to the copying of encrypted network packets from host to enclave. The higher standard deviation in TLS vs. noise is due to the packet size used in



(a)

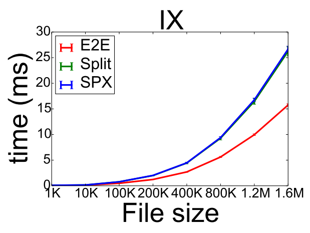


(b)

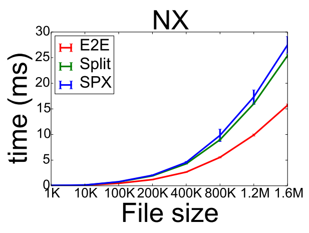
Figure 33: (a) Showing time time measured on client side in file transfer using TLX (SPX enabled TLS) over TCP; (b) Showing CDF of locally replayed web page load times for Alexa's top 100 web pages over TLX.

experiments. TLS experiments used blocks of 1KB for the file transfer where as Noise mandates the use of blocks of 65536 bytes. To put it in context, this means that with SPX a DPI solution for encrypted traffic in middle-boxes can be implemented as proposed in [142, 123], with a much lower overhead. Further, SPX approach also doesn't mandate apriori knowledge of what to look for in the encrypted traffic. In summary, SPX is a low overhead solution that provides full secure access under E^3 security properties.

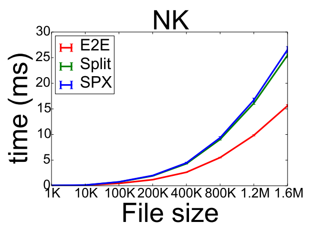
Web Page Loading. To understand how the SPX would perform in real-world performance, we examine web page load time. We recorded and replayed Alexa's top 100 websites using Mahimahi tool [125]. Figure 33(b) shows the cumulative distribution function (CDF) of the webpage load times with proxy only and with SPX enabled proxy. Evident from the graph is that there is modest SPX overhead in full web page loading. We attribute this to the structure of the web pages which include a large number of small objects, where SPX overhead is not amortized. However, important to consider the fact the we are locally replaying web pages over very fast networks. The web page loading which is usually in order of seconds is finished in milliseconds, thus presenting the worst case for TLX which is also evident on the right side in the figure. As the time to load increases, you can see that the performance gap diminishes.



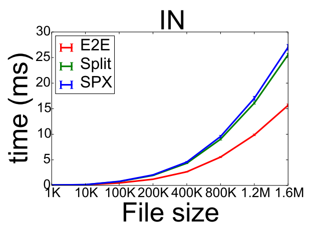
(a)



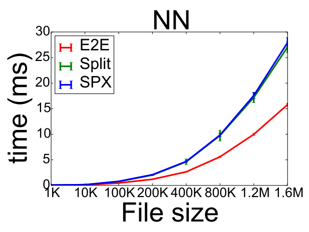
(b)



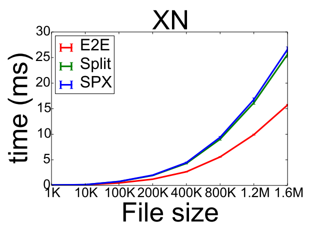
(c)



(d)



(e)



(f)

Figure 34: Showing time measured on client side in file transfer using different handshake patterns provided by NoiXE (SPX enabled Noise framework) using echo protocol over TCP.

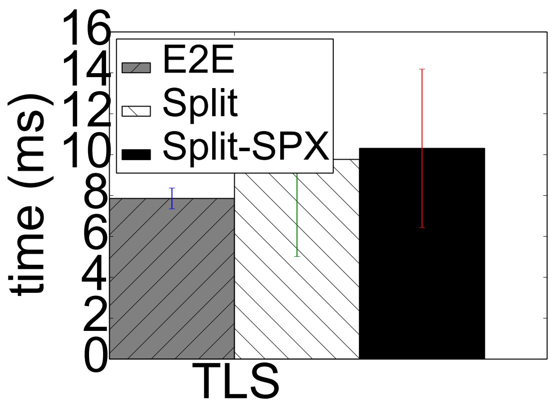
Table 18: Showing expressions for extra bytes introduced in handshake due to SPX.

Protocol	**Expression to calculate extra bytes in handshake**
Noise	attestation report size + bytes in one handshake + session key size
TLS	attestation report size + bytes in one handshake + certificate size + session key size

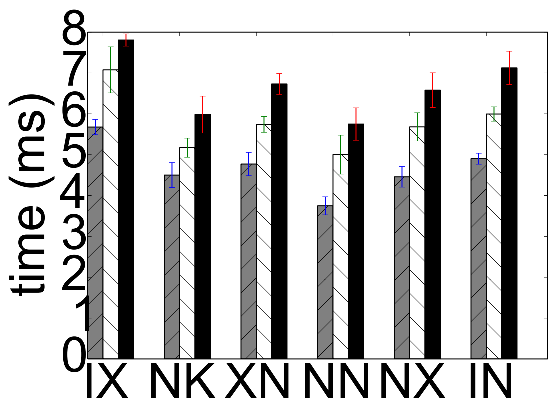
6.5.2 Micro Benchmarks

Handshake time. Figure 35 shows the time taken to carry out handshakes when client connects to server without any proxy (E2E), with a split connection proxy and finally, a SPX enabled proxy and server for TLS and 5 interactive handshake patterns. For readers, we list the handshake patterns in Noise terminology in Table 17. The results shown are averaged over 20 runs to show one standard deviation. We observed an overhead of only over our baseline. SPX approach outperforms the only solution reported in literature [142] that offers such security semantics. However, in offering such security semantics that work is limited in that it only allows a limited level of access to middle boxes.

Extra bytes in handshake. Since, the extra work is done at the edge and server side, end clients do not see any change in terms of number of bytes exchange to setting up a secure connection. Table 18 shows the expression to calculate the number of extra bytes due to SPX enabled edge functions and servers to setup a connection. It shows that the overhead in terms of extra bytes is reasonable SPX suitable for short-or long-lived connections. Compared to mcTLS [123] which requires multiple



(a)



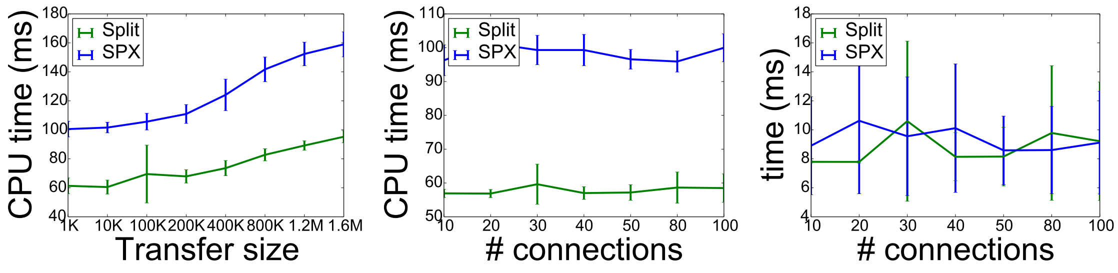
(b)

Figure 35: Comparing the handshake time of SPX enabled protocols (TLX, NoiXe) as observed on client side.

encryption keys for different contexts. This results in the handshake sizes proportional to number of contexts. Even if we do not consider the tediousness of explicitly creating multiple contexts in client traffic to service them at the edge, mcTLS can support a limited number of contexts due to size of the handshake. SPX enabled protocol allow edge function to fully access to traffic, in essence providing infinite contexts with constant overhead. Further, SPX provides E^3 security semantics where as mcTLS context available to a middlebox is insecure as that context can be decrypted and a malicious edge function with root privileges can steal that information. We only show results for TLS for further benchmarks. However, we report that we observed similar trends for Noise protocol.

Concurrent connections. Figure 36(c) show the handshake time measured on client side with number of concurrent connections. Evident from the figure is that SPX performance is not constrained by the number of concurrent connections. This is important for SPX to be a solution in edge computing scenarios because the edge infrastructure is by definition resource constraint. Further, it also highlights another point that with SPX edge functions can securely service multiple clients without being limited by the amount of memory addressable by SGX enclaves.

CPU Overhead. Figure 36 shows the average CPU consumption on the proxy and server during file transfer experiment. It clearly highlights that SPX puts substantial overhead 2x on CPU time during its operation. The consumption is due to memory copy of network packets from host to enclave and overhead due to memory page encryption. Compared to BlindBox [142] which uses much more CPU



(a) (b) (c)

Figure 36: Comparing CPU time spent on proxy side measured (a) during file transfer experiments; (b) multiple concurrent connections experiments; (c) Showing variation of time measured in file transfer using TLX with number of concurrent connections.

intensive homomorphic encryption and reduce functionality supported at the middle boxes deriving them from use cases, SPX approach provides full access to encrypted traffic with lower overheads. It is especially important for Edge computing perspective because this implies that developers have full flexibility to implement any logic for their edge functions.

6.6 SPX *Deployment scenarios*

In Mobile networks. We envision SPX to deliver a crucial functionalities as for edge computing applications described in ETSI's framework and reference architecture for Mobile Edge Computing [21]. SPX augments the static provisioning of certificates to edge functions with just in time remote attestation carried out by the EF itself to remove reliance on system level management to service end users. This is important to thwart efforts of spoofing an edge function to access important content. SPX with appropriate enhancements to interface with radio access network (RAN) equipment can be deployed in mobile networks.

In Enterprise. Recent trends suggest deployment of on-premise virtual customer premise edge (vCPE) equipment to reduce the number and cost of physical hardware appliances required for hosting value-added features (EFs). SPX can provide necessary support to allow edge functions deployed in those equipment to operate on encrypted traffic. We posit that vCPE providers can integrate SPX in a straight forward manner enabling them to benefit from end to end security guarantees. This can result in a better price-performance ratio for their deployed hardware.

Other scenarios. Although, this research is motivated by edge computing applications. similar approach can be utilized in other settings with appropriate modification which we believe will be trivial to carry out. For example, (i) to secure the control plane for the virtual network functions (VNF) - a direction all the mobile networks are moving towards, (ii) in inter data-centers specifically in hybrid cloud settings, only allowing appropriate access to relevant parties.

6.7 Chapter summary

We presented a design framework – SPX– to create secure protocol extensions to enable edge computing applications to operate over encrypted traffic without violating . SPX uses hardware assisted trusted execution environment such as Intel SGX for widely popular TLS and Noise based protocols. It is crucial for edge computing because edge infrastructure is by definition present in physically insecure locations. In creating SPX, we address a fundamental technical issue of binding the trusted execution with a communication channel that allows it to abide by the proposed E^3 semantics — secure semantics which are equivalent to end to end semantics but also explicitly recognizes the edge. In addition to outperforming state of the art, we considered difficult problems such as inter-operability, deployability in developing SPX to ensure that we put minimum constraints on developers of a emerging edge computing eco-system. SPX provides the example of innovative solutions that an attractive and sustainable edge computing eco-system would entail. More generally, it highlights an important point that existing abstractions e.g., end to end principle have to be revisited for their application in edge computing.

CHAPTER VII

RELATED WORK

We organize the related work with broader research areas that this thesis spanned.

7.1 Android App Delivery

Supporting apps from the Edge. Prior efforts have considered support app execution via edge-cloud platforms [106, 70, 100, 60] to leverage resource-rich execution environment to partially or fully offload app execution from resource constrained devices. But none of them considered use of edge cloud platforms for app delivery. In that sense, the research carried out as part of thesis is the first one.

Web traffic characterization. Previous work on characterizing Internet traffic workload has mainly focused on video e.g., YouTube access patterns [61, 158, 66, 107, 79] etc, and web but, there has been no work done in collecting the android app access patterns. This research carried out as part of this thesis is the first of its kind to capture and analyze the download behaviour of Android apps.

Application Streaming. Prior work on application streaming includes (i) 'thin' desktop clients [14, 30, 35, 42] streaming pixels to low cost front end displays. (ii) Numecent's cloud paging [37] streams memory pages to end clients. (iii) Instart logic's [25] web-app streaming exploits the order of relevance of app components using a browser-resident nano-visor to optimize web app delivery. A recent offering by Amazon performs mobile app streaming using a wrapper SDK. AppFlux differs from such efforts because (i) it focuses on mobile apps vs. desktop applications, and (ii) it exploits the emerging eBox tier.

Mobile app analysis. Earlier work has analyzed mobile apps from various perspectives, e.g., quality [97], unsafe exposure [84], security [156], privacy [150], etc. We provide a performance-centric analysis of mobile app structures relevant to enabling app streaming.

Online application update. Dynamic software updates [144, 124, 113] leading to reference management tools [48] have been explored for enterprise IT teams, with a focus on managing system updates for large organizations [7, 31]. But no such tools/technique exist to benefit end users of mobile devices and apps. AppFlux addresses this gap. Recent work to reduce mobile app update traffic proposes micro app updates [67] which would complement AppFlux.

Pre-fetching cache. Using dynamic caching for pre-fetching content, Gandhi et al [79] suggested that k-means clustering used more intervals while reducing error rate compared to dynamic programming. However, based on our android app access pattern, k-means clustering gave a cache-hit ratio of 75%, which is lower than the cache-hit ratio of the proposed p-LRU algorithm. Zink et al [158] observed a similar pattern for YouTube videos and proposed a caching policy based on LRU and popularity of a movie. The results showed an improvement in the cache-hit ratio. However, their algorithm depended on a global list of popular movies. Access to a global list may or may not be there. Also, it might happen that the global popularity list might differ from local lists [158], where they proved that there is no strong correlation is observed between global and local popularity and video clips of local interest have a high local popularity. The p-LRU algorithm addresses these issues, as it generates the popularity list by learning the access patterns locally.

7.2 *Edge infrastructure and edge function deployment*

The emergence of edge infrastructure is evident in research prototypes such as cloudlets [106] and eBoxes [57], and industry initiatives such as Microsoft's micro DCs [20] and ETSI Mobile Edge Computing [21]. Examples of edge cloud nodes include small cells [39, 8], WiFi routers [9, 40], or cloudlet servers [106], that could be located in homes or neighborhoods, stores, malls, offices, etc. ETSI MEC initiative is a nascent standardization effort with many industry participants. Although complete MEC implementations do not yet exist, the industrial push to define such architectures reflects a strong, shared belief in the need for edge infrastructure. Inherent to that is the need to timely consider the full spectrum of technologies that can address the requirements. AirBox is the first step towards those needs prior to the concrete definition of architecture which can get constrained by choosing a particular technology. The MEC white paper also enumerates security challenges faced by EFs, but relies on trusted platforms to address these. AirBox is a concrete first step towards a solution without strong assumptions of trust in the distributed platforms.

Edge function Provisioning. Use of virtual machines is proposed in conjunction with VM synthesis [87] capability to provision edge functions as a VM. This can be resource intensive and slow, despite optimizations such as caching base images and making available overlay files from backend. AirBox solves the same problem in a much more scalable fashion using Docker's capabilities. Jitsu [114] is proposed as a power-efficient and responsive platform for hosting cloud services in the edge network while preserving the strong isolation guarantees of a type-1 hypervisor but still relies on the hypervisor for security guarantees. Further, the Jitsu kernel based on the mini OS kernel may put additional porting effort and/or constraints on developers of EFs to create thin mirage OS based VMs for their EFs, and is susceptible to unavailability of standard tools to implement EFs. Paradrop [151] uses LXC containers to provide virtualization with significantly lower overheads which is similar to AirBox. LXC lack the seamless provisioning support provided by Docker and simple Docker file interface. In general, our research validates the idea of using containers for EFs by considering evaluating it for different capabilities of edge cloud platforms. Additionally, we analyze the upper bounds for provisioning activities. However, Paradrop does not address security aspect of EFs.

7.3 Edge Function Security

Edge function security has drawn less attention alongside the fast growth of edge cloud research. Several surveys[140, 96, 145, 133] have pointed out security and privacy challenges from various perspectives. For example, how to ensure the confidentiality, integrity, and availability of the data in edge cloud; how to prevent edge cloud services against external attacks (man-in-the-middle attack) and internal attacks (compromised environment). Possible mitigation techniques are also proposed for different kinds of threats. However, there is no existing platform that takes various security considerations into account. AirBox is proposed to be a practical edge cloud platform that attempts to address existing security challenges by integrating the novel commodity CPU features (Intel SGX). AirBox [59] proposed to attempt the address handling of secure traffic in edge services by integrating Intel SGX. But the approach was not protocol centric and did not address the limitations of TEE hardware. SPX fills that gap.

7.4 Handling end-to-end encrypted communication

Protocol based solutions. Earlier approaches to provide access to optimizing web proxy proposed to create split connections [109] at SSL layer but in essence required violating end to end principles. More recently, [123] proposed the use of multi-context encryption scheme as extension of TLS protocol. It involves encrypting the data, or part of the data, with different keys (or contexts). Then, selectively sharing those keys with middleboxes to allow them access to pre-defined part of the traffic. It shares keys to partial content, it makes the traffic more prone to side channel attacks and weakens the end to end semantics. Another approach, [142] suggests the use of homomorphic encryption to allow introspection in the middleboxes with notion of use case defined privacy constructs. However, homomorphic based encryption computation is far from practical in terms of its computational requirements to be a viable option for edge computing applications. Finally, none of the above mentioned works can handle privileged software attacks [64].

Secure communication channel. Under a threat model where host is untrusted, some works focus on building a secure communication channel on the same machine. Zhou et al. [157] outline an approach to secure the data-transfer between user's I/O device and a program trusted by the user. Jang et al. [99] propose SeCReT, a framework to build a secure channel between TEE and non-TEE. SPX targets on establishing secure end-to-end channel over the Internet thus is orthogonal to these works. Further, compared to SPX approach of binding TLS session with remote attestation on the server side, another approach is to replace the TLS certificate by the Attestation Identity Key (AIK) certificate. However, this solution is not suitable for multi-tenant virtualized edge computing scenario because: (i) AIK certificate is typically unique for one host machine making it vulnerable for MITM attack, (ii) cost associated with applying a TLS certificate if we consider assigning a TLS certificate to each edge function and (iii) performance overheads associated with execution in TEE as we would have to use it twice for every new session: one for decrypting the pre-master secret and one for generating the attestation.

Trusted execution environments. Other direction in research has focused on providing a trusted execution environment (TEE) by reducing the trusted entities to a small verifiable code base [51, 65, 90, 146, 152, 155, 132] and trusting only the hardware [52, 126, 138]; they lack discussion on how to enable establish trustworthiness of communication channels with such primitives, which is an inseparable and critical part of edge computing.

TEE based remote attestation. Several works make use of remote attestation to provide trustworthiness of the local TCB to a remote party. Flicker [118] provides hardware-supported isolation for executing sensitive code. By using TPM, the executed code can be attested by a remote party. Due to significant performance overhead introduced by the design of Flicker, a follow-up work, TrustVisor [117], achieves the similar security guarantee by implementing the system in the hypervisor level. The remote attestation capability is still supported by leverging TPM. Cloud Terminal [115] also adopts TPM-based attestation which allows the server to validate the client application. Haven [52] leverages Intel SGX to prevent code and data from tampering by an untrusted cloud provider. SGX-based attestation is used to provide a proof of the program is correctly loaded and executes in an SGX enclave to a cloud customer. The above-mentioned works are the concrete examples of using remote attestation. However, these works either fail to provide detail on the how to perform remote attestation protocol or only consider the naive protocol. SPX can be used to fill the gap in the protocol part.

Other forms of remote attestations. The original version of remote attestation mechanism proposed by TCG has suffered from several limitations e.g., Direct Anonymous Attestation [62] EPID [63], Semantic remote attestation [88] Property-based attestation [134]. Although, the design described in this paper utilizes Intel SGX support for remote attestation, SPX offers high flexibility to adopt any type of remote attestation mechanism as long as it provides the requires OS agnosticism.

7.5 Concepts and System Mechanisms

Ephemerality. Lacuna [73] propose the idea of secure ephemeral channels to guarantee forensic deniability. Similar to Lacuna's principle, for app slices, app components and state are erased but without any strong deniability guarantees. Ephemeral community 4chan [54] is known for ephemeral content streaming at scale, compared to that AppSlice can be seen as stream of ephemeral apps.

Program Slicing A large body of work has focused on program slicing [92]. The primary focus of many such efforts has been to identify program execution components and to apply those findings for predicting performance or resource consumption [108, 110]. We apply a similar methodology of analyzing the app components' dependency graph in packaging activities in individual app slices. Additional opportunities exist in further leveraging prior work on program analysis to further optimize the slicing process.

End-to-edge-to-end (E^3) Security Semantics The end-to-end system design principle was first introduced in [135] which has been applied to networked systems and to encryption in TLS to provide confidentiality. Active networks [147] proposed to change those semantics but without much practical success. However, ubiquitous presence of middleboxes suggests otherwise. Despite that there remains a lack of concepts to reason about their security semantics. (E^3) Security Semantics fills that gap.

CHAPTER VIII

FUTURE DIRECTIONS

Edge computing will pave the way for emerging Internet applications. The insights gathered through this dissertation work suggest several promising research directions.

8.1 Towards a richer app ecosystem

There are a number of opportunities in existing mobile app eco-system that can create a more compelling case for apps in future. We outline some of those below which can be considered as being opened up by work in this thesis.

8.1.1 Semantic gap in app functionalities and their use

With AppInstant, we demonstrated how the future models of end user interactions can benefit from the large momentum of app eco-system. However, a problem that remains open is mapping the semantic knowledge of the functionality provided by an app vs. with what user wants to do i.e., user intent. In our earlier work [60] (not included as part of this thesis), we proposed the abstract notions of competence, Intent and Context for edge computing applications. We see opportunities especially in emerging assistant like devices at the edge e.g., Amazon echo, Google Home, etc., of applying those abstractions and technologies developed as part of this thesis. More generally, mapping user intent to the functions provided by a device is an interesting area with potential of changing the dynamics of how end users interact with devices and web services.

8.1.2 On-demand, contextual management of Internet of Things

The Internet of Things (IoT) presents a promise of a future with billions of interconnected devices, where end user interactions with such devices, or services enabled by device-device interactions augment our cognitive abilities and improve our lives‘ experiences. Achieving this requires that interactions with and across such things are effortless, regardless of whether the environments where we rely on IoT services are ones we repeatedly inhabit (like our homes, offices, vehicles, or neighborhood stores), or ones that are seldom, if ever, revisited (like conference venues, or remote airports and hotels). Yet, there are daunting practical problems with achieving this goal. In the absence of standards, establishing a universal lingua franca to communicate with or across devices, is not realistic. One solution to this can involve dynamic download and installation of device-specific applications, but given current technologies and the current state of the mobile app ecosystem, these explicit *download/install* related steps affect the seamlessness of the IoT experience. More generally, research in the direction that allows decoupling of the IoT device interface from the device itself and making it available dynamically and on-demand based on context can lead to high impact outcomes.

8.2 Open problems in edge computing

Mobile edge computing is posed as an integral part of 5G expected to hit the mainstream [21, 20, 137, 60] by 2020. More importantly, the promise of new revenue generating ecosystem makes it strategically important for mobile network operators and enterprises. There exists a plethora of research opportunities that could result in high impact outcomes. We list a few below:

8.2.1 Capacity planning and data placement in edge computing

The critical challenges for a web service to utilize the mobile egde data centers are determining their appropriate locations, deployment of its edge functions and placement of associated data at those mobile edge data centers. The problem of data placement among geographically distributed data centers has motivated sophisticated approaches (Volley [49], Acorn [154]), etc.

For mobile edge data centers, the problem becomes more difficult and hence, requires a new approach because of the following reasons:

- *High degree to distribution:* the number of mobile edge data centers is expected to be in order of 100,000s as compared to 10s or 100s for core data centers.
- *Shorter decision window:* The resource limitations and user mobility leads to a shorter time window to decide and execute data placements. A more aggressive and pragmatic approach is needed for data placements at the mobile edge.
- *Potentially higher WAN costs:* Although, we are not aware of any price differentiation between WAN traffic and the traffic that would be placed at the edge but we posit that the cost of sending

a data to the mobile edge which would require traveling via mobile back-haul may be costlier than over conventional IP network.

- *Severely limited resources:* The limited availability of resources at the edge data centers stems from the allowed power levels, heating concerns arising out of safety regulations and availability of real estate to create those data centers. A typical mobile edge data center is expected to be something like a half or full 42U rack of servers.

Addressing each of these factors can lead to interesting research results that can help shape the next generation Internet eco-system.

8.2.2 Addressing Mobility for edge functions

Concretely, user mobility which motivated earlier work in geo-replication or data placement considered user mobility between data centers. However, the impact of user mobility among edge data centers is much more pronounced because of the smaller area an edge data center can service e.g., one located at a GSM cellular tower can service clients in its range i.e., 70 KM. At first glance, it seems like a simple problem that is solved by mobile networks and virtual machine migration in data center world. However, the existing approaches are either too heavy to be considered for edge functions (e.g., VM migration) because of the inherent assumptions they make i.e., trust the infrastructural elements i.e., cellular tower among with hand off is carried out or availability of a common shared file system which is not true for edge functions. So, that problem remains open. Further, the problem becomes even more difficult and hence, requiring a fresh approach, if we consider migration of secure edge functions i.e., running inside secure SGX enclaves which are inherently tied to a single processor on which it is running. More generally, the direction of addressing mobility for edge computing applications remains an open and interesting research direction.

8.2.3 Formalizing problems in edge computing

The capacity planning, data placement and mobility problems described above are fundamentally are NP complete problems. We posit that novel ways to formalize edge computing eco-system as a whole can lead to fresh approach to solving this difficult problems. As a first concrete step, we present the following formulation which can be used as starting point towards that:

- *Infrastructure map (i-map)* can be thought of as a static input to system containing locations of edge data centers,
- *Access map (a-map)* can be thought of as a dynamic input containing spatial and temporal characteristics of end user accesses to the service,
- *Edge map (e-map)* can be thought of as intermediate output structure containing the selected edge data centers where edge functions and state must be deployed while being inside some constraints e.g., a the cost envelop,
- *Data map (d-map)* can be thought as the final output which can be fed in to data transfer utility that can actually carry out deployment and/or move the data.

A web service can make use of this formulation by carrying out the following actions.

- *Preparation Phase.* Before running a jobs a web service creates the i-map from the information about the mobile edge infrastructure, a-map using the access patterns of its users and their locations.
- *Selection Phase.* Then, using input i-map and a-map as input, a web service can select appropriate mobile edge data centers that fit in to its cost envelop. This is first instance where we can reduces the number of locations to be considered. Fundamentally, this is a subset sum problem which is exponential with respect to number of mobile edge data centers.
- *Placement phase.* Then, it can assign mobile edge data centers to one of the regional data centers OR CDNs (macro edge). At the macro edge, using appropriate a-map partitions – data placement can be calculated thus, providing a device and conquer approach. Fundamentally, this is a clustering operation. The intuition behind it is that due to storage constraints at mobile edge data centers even if we calculate exactly, we may not be able to benefit from it.

However, This was just an example of formulation. We believe that there exists research opportunities in more elaborate and elegant formulations that can lead to development of practical solutions.

8.2.4 Exploration of new system level concepts

E^3 semantics and ephemerality presented in this thesis provide examples of new abstract concepts that a attractive and sustainable edge computing eco-system would entail. There are many interesting assumptions in networking and systems that may be violated or not required in edge computing scenarios. This opens up opportunities to develop those new concepts and position them to solve difficult problems.

8.3 Chapter Summary

In this chapter, we outlined concrete research directions and open problems in Edge computing space. It is our hope that this is useful for a young researcher or research community in general to provide an concrete set of directions to explore in their research. We hope that of the contributions made by this thesis can pave the way for future approaches and research to address technical challenges to assimilate edge computing solutions in to main stream computing eco-system.

CHAPTER IX

CONCLUSIONS

This thesis contributes systems and concepts that address technical challenges associated with assimilation of edge computing in general computing landscape. Towards that, we introduced the notion of edge functions as application specific logic running on edge computing infrastructure.

9.1 Summary

In the first part of thesis, we presented concrete edge function use cases that address real life problems faced by Android app eco-system. We also presented the first of its kind measurement of app delivery traffic and a rigorous analysis of Android apps. This allowed us to gain unique insights into the Android app eco-system. Firstly, it allowed us to quantify the pain points in app delivery: back-haul bandwidth use in the last mile, app discovery and relevancy in emerging models of user interaction with web services. Further, it highlighted the need for developing new low overhead system level concepts – ephemerality – in Mobile OSes that can unlock the full potential of edge functions. Addressing those concerns, we presented four concrete systems: (i) AppFlux, (ii) AppSachet, (iii) Ephemeral apps and (iv) AppInstant which demonstrated the benefits the edge functions can deliver to highly dynamic and large scale Android app ecosystem. We demonstrated that we are able to achieve upto 83% lower back-haul bandwidth usage in the last mile which cannot be addressed by existing computing infrastructure while ensuring atleast 2x faster app delivery at a reasonable cost and leveraging the proposed system software support for app streaming based app updates. Further, using the proposed new ephemeral app model made possible by system support for ephemerality, we demonstrated that we are able to reduce barriers to app discovery by 10x without compromising of performance or requiring any changes for end users or app developers. Finally, we demonstrated that with AppInstant how edge functions based app slicing functionality along with earlier discussed system support can bridge the gap between existing (apps) and emerging (messaging) model of web service consumption without a large scale developer mobilization for another computing eco-system.

This highlights a concrete instance of transformative change in a large scale computing eco-system made possible by edge functions. More generally, we highlighted the importance and benefits of employing application specific logic embedded in and as edge functions, justifying their definition and deployment of edge computing infrastructure.

In second part of the thesis, we presented our research to translate those benefits to arbitrary web services. Towards that we presented a model of generic edge functions allowing us to reason about their benefits and led to identification of the ABACUS edge functions – that exemplify the characteristics of different edge functions and lower the constraints on researchers paving a smoother way to reason about system level functionalities for edge functions. We demonstrated that by deriving the technical challenges for edge computing based on the lessons learnt in developing the above mentioned edge functions. For edge functions to seamlessly assimilate in to mainstream computing ecosystem we identified the following challenges: (i) fast, scalable and secure provisioning, (ii) low developer constraints in developing them and (iii) their secure execution and confidentiality guarantees for their state stored at untrusted edge infrastructure; . In

In third part of thesis, we presented the quantitative exploration of state-of-the-art technology including virtual machines, containers and application sandboxes carried out to identify suitable candidate for addressing the above mentioned challenges in bootstrapping the edge function ecosystem. In doing this, we rationale was to profess a more scientific approach to choice of the technology for a nascent edge computing eco-system rather than popularity. Based on our analysis, we presented a Docker based edge function platform – AirBox – developed as part of this thesis. AirBox enables fast, scalable and secure deployment of edge functions in arbitrary edge infrastructure. For AirBox edge functions, we prescribed the use of state-of-the-art security features in processors i.e., Intel SGX to address their security concerns and those associated with the use of container technology. Further, to reduce developer effort in using Intel SGX, we presented an intuitive and general interface for edge function developers that simplifies the non-trivial use of Intel SGX in their edge functions. Concretely, we demonstrated that AirBox outperformed competing solutions by 10x in deployment performance and is demonstrated to provided highest level of system security for edge functions, bar none. To the best of our knowledge, we were the first one to propose the use of Docker containers and Intel SGX together.

In the fourth part of this thesis, we presented a solution for the the most critical problem faced by edge computing which could render the use of whole paradigm impractical. The problem of enabling edge functions to operate on encrypted traffic without compromising end-to-end security semantic and without adverse effects on their performance. We presented an protocol level approach – Secure protocol extensions (SPX) – to develop protocol extensions for secure protocols that guarantees the security properties of a protocol while at the same time allows edge functions to operate on encrypted traffic using OS agnostic trusted execution environment such as those provided by Intel SGX. We demonstrated its applicability for TLS and Noise based protocols expected to gain popularity in emerging IoT scenarios. Further, we demonstrated that we are able to achieve the goals with mere 12-15% overhead, which gets amortized for real world applications such as browsing and file transfer. As part of this work, we also introduced the notion of end-to-edge-to-end E^3 security semantics. However, we argue that similar to end-to-end argument in system design, the E^3 semantics are more practical and general system level notions that can be applied to other aspects of systems as well.

9.2 Lessons

We learned a few important and un-intuitive lessons while working on this thesis discussed below.

9.2.1 Edge computing $\neq$ Minified Cloud computing

A high level learning from the thesis is that edge computing must not be treated as simply application of the technologies developed for cloud computing paradigm at a small scale. Intuitively, it is a very common approach and is visible in a number of research projects and solutions proposed for edge computing. For instance, the availability of elastic and practically unlimited resources for services in cloud scenarios is not valid for edge infrastructure, which by definition is limited in resources. The availability of pre-provisioned network between different nodes of the cloud is something that is simply not true for edge nodes placed in distributed locations. Similarly, the assumed physical security of compute infrastructure is violated for edge.

As concrete example of those assumptions, we observed that the use of virtual machines to provide deployment and multi-tenancy support in edge computing has been assumed without understanding the trade-offs in using other technologies. However, for edge computing it is crucial to consider per node efficiency and concurrent provisioning performance which was not even considered in other solutions as problems. This is exactly what we demonstrated with AirBox project. Similarly, addressing the assumptions about secure communication between different edge nodes or between clients-edge on behalf of a web-service led us to develop secure protocol extensions (SPX).

Additionally, there are inherent un-intuitive constraints in using the potential technologies, such as developer constraints, experienced while developing edge function for Android app eco-system, in making Android specific changes, using specific toolchains, etc. Constraint like those should also be considered when realizing a sustainable edge function eco-system. It is these non-intuitive constraints and requirements that can lead to unique solutions developed as part of the edge computing space.

9.2.2 Potential for transformational changes

A common view of edge computing is to consider edge functions as supporting functionality (load balancer, proxy etc.), of back-end web services. But we observed that real potential of edge functions is unlocked when the systems and approaches are designed with explicit consideration of the edge infrastructure as part of the computing eco-system.

An example demonstrated as part of thesis is the notion of ephemeral apps for Android app ecosystem which can potentially lead to transformational changes in how apps are delivered, used and can also be integrated in emerging models of web service consumption. Another example of this manifested itself in form of the cache policies and cost model for app caches developed as part of AppSachet which highlights a simple fact that the benefits of edge functions can be realized with minimal costs. We learnt that it is an important to vet all existing technologies when using them in edge computing systems.

We hope that this transformational angle of edge computing is kept in mind by future researchers and/or application developers to create edge native solutions/applications. We hope that it is considered in the way future research is carried out in edge computing space leading to much more compelling use cases and systems.

9.2.3 Need for new abstractions

A common approach to developing practical systems is to stick with existing system level abstractions and mechanisms to develop solutions for and/or based on edge computing.

For example, the well regarded end-to-end principle, specifically as applied to network transport encryption, had to be re-visited to enable edge functions to operate on end-to-end encrypted traffic transported using secure protocols without compromising their security semantics. In developing the secure protocol extensions (SPX) approach we defined a new End-to-Edge-to-End (E^3) security semantics — a equivalent of end-to-end semantics but with explicit recognition of the edge component in eco-system. Further, we observed that newer formulations are needed to characterize benefits of edge functions depending on where measurements are carried out e.g., at client side, at the edge or at the remote cloud. This led us to develop different expressions for latency and bandwidth benefits.

These experiences lead us to believe that new system level approaches and abstractions may be required to explicitly recognize the edge infrastructure as part of computing ecosystem and harness the full potential of edge functions.

9.3 Thesis Concluding Statement

This thesis provides hard evidence that edge functions can deliver latency reduction and bandwidth consolidation for current device-cloud interactions. Not only that but it shows that edge functions can lead to transformational changes in the computing ecosystem. Solutions developed as part of this thesis address the critical problems and highlight the technology trade-offs in the mechanisms for edge computing and for a practical edge function eco-system. However, there remain interesting open problems, trade-offs to be explored and new solutions to be developed for the edge function eco-system, providing a fertile ground for systems, networking and security research.

REFERENCES

[1] "Android app usage api @ https://goo.gl/39dqlu."

[2] "Android apps per device - yahoon avaite @ http://goo.gl/3zb3ob."

[3] "Android apps per device @ http://goo.gl/zkaxsx."

[4] "Android content provider." `http://goo.gl/o1CX3h`.

[5] "Android debug bridge.." `http://goo.gl/Mzvl3j`.

[6] "Android graphics - https://goo.gl/iuu5vq."

[7] "Apple enterprise management tool @ https://goo.gl/isvycs."

[8] "Att small cell deployment plans." `http://goo.gl/XdfkfH`.

[9] "Att wifi hotspot locations." `http://goo.gl/0rs2LZ`.

[10] "Aufs - layered filesystem @ http://goo.gl/z5l9fc."

[11] "Bionic: Android's libc documentation in ndk docs.." `http://goo.gl/e977ST`.

[12] "Cdn pricing 2014– http://goo.gl/7l7fkd."

[13] "Cisco visual networking index: Global mobile data traffic forecast update, 2015âĂŞ2020 white paper." http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html.

[14] "Citrix xenapp @ http://www.citrix.com/products/xenapp/how-it-works/application-virtualization.html."

[15] "Dell edge iot gateway." http://www.dell.com/us/business/p/edge-gateway.

[16] "Dex2jar tool @ https://code.google.com/p/dex2jar/."

[17] "Docker." `https://www.docker.com/`.

[18] "Docker on windows." `http://goo.gl/oAT5NM`.

[19] "Docker trusted registry service." `https://goo.gl/lTZWhM`.

[20] "Emergence of micro data centers at the edge." `http://research.microsoft.com/en-us/um/people/bahl/Present/Bahl_mDC_Keynote_May.pdf`.

[21] "Etsi mobile edge computing." `http://goo.gl/Qef61X`.

[22] "Global internet phemomena spotlight - encrypted traffic." `http://goo.gl/KP0d0n`.

[23] "Google saved game service." `http://goo.gl/k2fVuB`.

[24] "Google streaming app content - http://goo.gl/mm5hvi."

[25] "Instart logic web application streaming @ http://instartlogic.com/."

[26] "IntelÂő software guard extensions programming reference." `https://goo.gl/sy0tRu`.

[27] "Internet rush hour." `http://goo.gl/gvaS7M`.

[28] "Level 3 cdn reports last mile as new bottleneck @ http://goo.gl/3ir9kg."

[29] "Make your network edge intelligent and meet tomorrowâĂŹs needs today - cisco white paper." http://www.cisco.com/c/dam/en/us/solutions/collateral/enterprise-networks/unified-access/network-edge.pdf.

[30] "Microsoft appv @ http://www.microsoft.com/en-us/windows/enterprise/products-and-technologies/mdop/app-v.aspx."

[31] "Microsoft system center @ http://goo.gl/uhfiev."

[32] "Mobile apps overtake PC Internet usage in U.S. - CNN Money." `http://goo.gl/qd1vo4`.

[33] "Nest development platform."

[34] "Netflix open connect." https://openconnect.netflix.com/en/.

[35] "Novell application virtualization @ http://www.novell.com/products/zenworks/applicationvirtualization."

[36] "Nsa-linked cisco exploit poses bigger threat than previously thought." http://arstechnica.com/security/2016/08/nsa-linked-cisco-exploit-poses-bigger-threat-than-previously-thought/.

[37] "Numecent cloud paging @ http://goo.gl/k34n4d."

[38] "Open edge computing." http://openedgecomputing.org/about-oec.html.

[39] "Qualcomm small cells." `http://goo.gl/HEpudP`.

[40] "Qualcomm smart gateways." `http://goo.gl/BwPc7f`.

[41] "Substantial licensed spectrum deficit (2015-2019): Updating the fccâĂŹs mobile data demand projections." http://www.ctia.org/docs/default-source/default-document-library/brattle_-350MHz_licensed_spectrum.pdf.

[42] "Symantec workspace streaming @ http://www.symantec.com/workspace-streaming."

[43] "Trends and stats about web - http archive - http://httparchive.org/."

[44] "Understanding 5g: Perspectives on future technological advancements in mobile." `https://www.gsmaintelligence.com/research/?file=c88a32b3c59a11944a9c4e544fee7770`.

[45] "An updated performance comparison of virtual machines and linux containers." `http://goo.gl/fhDwse`.

[46] "Webpagetest speed index metric - http://goo.gl/rw3d5d."

[47] "Whatsapp encryption overview - technical whitepaper." `https://goo.gl/MuHvjC`.

[48] "Wikipedia - reference management tools @ http://goo.gl/c7pbha."

[49] AGARWAL, S., DUNAGAN, J., JAIN, N., SAROIU, S., WOLMAN, A., and BHOGAN, H., "Volley: Automated data placement for geo-distributed cloud services," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, (Berkeley, CA, USA), pp. 2–2, USENIX Association, 2010.

[50] AMAZON, "Amazon web services (aws) - cloud computing services." `http://aws.amazon.com/`.

[51] AMAZON WEB SERVICES, "AWS CloudHSM," 2013. `http://http://aws.amazon.com/cloudhsm/`.

[52] BAUMANN, A., PEINADO, M., and HUNT, G., "Shielding applications from an untrusted cloud with haven," pp. 267–283.

[53] BAUMANN, A., PEINADO, M., and HUNT, G., "Shielding applications from an untrusted cloud with haven," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, (Berkeley, CA, USA), pp. 267–283, USENIX Association, 2014.

[54] BERNSTEIN, M. S., MONROY-HERNÁNDEZ, A., HARRY, D., ANDRÉ, P., PANOVICH, K., and VARGAS, G. G., "4chan and/b: An analysis of anonymity and ephemerality in a large online community.," in *ICWSM*, 2011.

[55] BHARDWAJ, K., AGARWAL, P., GAVRILOVSKA, A., and SCHWAN, K., "Appsachet: Distributed app delivery from the edge cloud," in *MobiCASE 2015*.

[56] BHARDWAJ, K., AGARWAL, P., GAVRILOVSKA, A., and SCHWAN, K., "Appsachet: Distributed app delivery from the edge cloud," in *MobiCASE 2015*.

[57] BHARDWAJ, K., AGARWAL, P., GAVRILOVSKA, A., SCHWAN, K., and ALLRED, A., "Appflux: Taming mobile app delivery via app streaming," in *TRIOS 15*.

[58] BHARDWAJ, K., AGARWAL, P., GAVRILOVSKA, A., SCHWAN, K., and ALLRED, A., "Appflux: Taming mobile app delivery via streaming," in *TRIOS '15*, (Monterey, CA, USA), USENIX Association.

[59] BHARDWAJ, K., SHIH, M., AGARWAL, P., GAVRILOVSKA, A., KIM, T., and SCHWAN, K., "Fast, scalable and secure onloading of edge functions using airbox," in *Proceedings of first IEEE/ACM symposium on Edge Computing*.

[60] BHARDWAJ, K., SREEPATHY, S., GAVRILOVSKA, A., and SCHWAN, K., "Ecc: Edge cloud composites," in *Proceedings of the 2014 2Nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, MOBILECLOUD '14, (Washington, DC, USA), pp. 38–47, IEEE Computer Society, 2014.

[61] BRAUN, L., KLEIN, A., CARLE, G., REISER, H., and EISL, J., "Analyzing caching benefits for youtube traffic in edge networks x2014; a measurement-based evaluation," in *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pp. 311–318, April 2012.

[62] BRICKELL, E., CAMENISCH, J., and CHEN, L., "Direct anonymous attestation," pp. 132–145.

[63] BRICKELL, E. and LI, J., "Enhanced privacy id: A direct anonymous attestation scheme with enhanced revocation capabilities," in *Proceedings of the 2007 ACM workshop on Privacy in electronic society*, pp. 21–30, 2007.

[64] CHECKOWAY, S. and SHACHAM, H., "Iago attacks: Why the system call api is a bad untrusted rpc interface," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, (New York, NY, USA), pp. 253–264, ACM, 2013.

[65] CHEN, X., GARFINKEL, T., LEWIS, E. C., SUBRAHMANYAM, P., WALDSPURGER, C. A., BONEH, D., DWOSKIN, J., and PORTS, D. R., "Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 2–13.

[66] CHENG, X., "Understanding the characteristics of internet short video sharing: Youtube as a case study," in *Procs of the 7th ACM SIGCOMM Conference on Internet Measurement, San Diego (CA, USA), 15*, p. 28, 2007.

[67] CHEUNG, A., RAVINDRANATH, L., WU, E., MADDEN, S., and BALAKRISHNAN, H., "Mobile applications need targeted micro-updates."

[68] CHUN, B.-G., IHM, S., MANIATIS, P., NAIK, M., and PATTI, A., "Clonecloud: elastic execution between mobile device and cloud," in *Eurosys '11*.

[69] CUERVO, E., BALASUBRAMANIAN, A., CHO, D.-K., WOLMAN, A., SAROIU, S., CHANDRA, R., and BAHL, P., "Maui: making smartphones last longer with code offload," in *Mobisys '10*, ACM.

[70] DIXON, C., MAHAJAN, R., AGARWAL, S., BRUSH, A., LEE, B., SAROIU, S., and BAHL, P., "An operating system for the home," in *NSDI*, USENIX, April 2012.

[71] DOUCEUR, J. R., ELSON, J., HOWELL, J., and LORCH, J. R., "Leveraging legacy code to deploy desktop applications on the web," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, (Berkeley, CA, USA), pp. 339–354, USENIX Association, 2008.

[72] DUNN, A. M., LEE, M. Z., JANA, S., KIM, S., SILBERSTEIN, M., XU, Y., SHMATIKOV, V., and WITCHEL, E., "Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, (Hollywood, CA), pp. 61–75, USENIX, 2012.

[73] DUNN, A. M., LEE, M. Z., JANA, S., KIM, S., SILBERSTEIN, M., XU, Y., SHMATIKOV, V., and WITCHEL, E., "Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels.," in *OSDI*, pp. 61–75, 2012.

[74] ELECTRIC, S., "The drivers and benefits of edge computing." http://it-resource.schneider-electric.com/i/637656-wp-226-the-drivers-and-benefits-of-edge-computing.

[75] EVANS, C., PALMER, C., and SLEEVI, R., "Public key pinning extension for http." `http://tools.ietf.org/html/rfc7469`, 2015.

[76] "Facebook Messagener Platform." https://developers.facebook.com/docs/messenger-platform.

[77] "Apps Solidify Leadership Six Years into the Mobile Revolution - Flurry." `http://goo.gl/GdztYX`.

[78] FRAHIM, J., PIGNATARO, C., APCAR, J., and MORROW, M., "Cisco: Securing the internet of things: A proposed framework." `http://www.cisco.com/c/en/us/about/security-center/secure-iot-proposed-framework.html`.

[79] GANDHI, A., CHEN, Y., GMACH, D., ARLITT, M., and MARWAH, M., "Minimizing data center sla violations and power consumption via hybrid resource provisioning," in *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC '11, (Washington, DC, USA), pp. 1–8, IEEE Computer Society, 2011.

[80] GARRISS, S., CÁCERES, R., BERGER, S., SAILER, R., VAN DOORN, L., and ZHANG, X., "Trustworthy and personalized computing on public kiosks," in *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, (New York, NY, USA), pp. 199–210, ACM, 2008.

[81] "Googlebot." https://www.google.com/webmasters/.

[82] GORDON, M. S., JAMSHIDI, D. A., MAHLKE, S. A., MAO, Z. M., and CHEN, X., "Comet: Code offload by migrating execution transparently.," in *OSDI '12*.

[83] GOYAL, S. and CARTER, J., "A lightweight secure cyber foraging infrastructure for resource-constrained devices," in *Proceedings of the Sixth IEEE Workshop on Mobile Computing Systems and Applications*, WMCSA '04, (Washington, DC, USA), pp. 186–195, IEEE Computer Society, 2004.

[84] GRACE, M. C., ZHOU, W., JIANG, X., and SADEGHI, A.-R., "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, (New York, NY, USA), pp. 101–112, ACM, 2012.

[85] HA, K., CHEN, Z., HU, W., RICHTER, W., PILLAI, P., and SATYANARAYANAN, M., "Towards wearable cognitive assistance," in *Mobisys '14*.

[86] HA, K., CHEN, Z., HU, W., RICHTER, W., PILLAI, P., and SATYANARAYANAN, M., "Towards wearable cognitive assistance," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, (New York, NY, USA), pp. 68–81, ACM, 2014.

[87] HA, K., PILLAI, P., RICHTER, W., ABE, Y., and SATYANARAYANAN, M., "Just-in-time provisioning for cyber foraging," in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, (New York, NY, USA), pp. 153–166, ACM, 2013.

[88] HALDAR, V., CHANDRA, D., and FRANZ, M., "Semantic remote attestation: a virtual machine directed approach to trusted computing," in *Proceedings of Usenix Virtual Machine Research and Technology Symposium (vol. 2004)*, 2004.

[89] HOEKSTRA, M., LAL, R., PAPPACHAN, P., PHEGADE, V., and DEL CUVILLO, J., "Using innovative instructions to create trustworthy software solutions," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, (Tel-Aviv, Israel), pp. 1–8, 2013.

[90] HOFMANN, O. S., KIM, S., DUNN, A. M., LEE, M. Z., and WITCHEL, E., "Inktag: secure applications on an untrusted operating system,"

[91] HOFMANN, O. S., KIM, S., DUNN, A. M., LEE, M. Z., and WITCHEL, E., "Inktag: Secure applications on an untrusted operating system," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, (New York, NY, USA), pp. 265–278, ACM, 2013.

[92] HORWITZ, S., REPS, T., and BINKLEY, D., "Interprocedural Slicing Using Dependence Graphs," in *Proc. of Programming Languages Design and Implementation (PLDI)*, 1988.

[93] HOWELL, J., ELSON, J., PARNO, B., and DOUCEUR, J. R., "Missive: Fast application launch from an untrusted buffer cache," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, (Berkeley, CA, USA), pp. 145–156, USENIX Association, 2014.

[94] HOWELL, J., PARNO, B., and DOUCEUR, J. R., "Embassies: Radically refactoring the web," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, (Berkeley, CA, USA), pp. 529–546, USENIX Association, 2013.

[95] HOWELL, J., PARNO, B., and DOUCEUR, J. R., "How to run posix apps in a minimal picoprocess," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, (Berkeley, CA, USA), pp. 321–332, USENIX Association, 2013.

[96] ILLERA, R., ORTEGA, S., and PETYCHAKIS, M., "Security and privacy considerations for cloud-based services and cloudlets," 2013.

[97] IVAN, I. and ZAMFIROIU, A., "Quality analysis of mobile applications,"

[98] JAIN, P., DESAI, S., KIM, S., SHIH, M.-W., LEE, J., CHOI, C., SHIN, Y., KIM, T., KANG, B. B., and HAN, D., "OpenSGX: An Open Platform for SGX Research," in *Proceedings of the Network and Distributed System Security Symposium*, (San Diego, CA), Feb. 2016.

[99] JANG, J., KONG, S., KIM, M., KIM, D., and KANG, B. B., "Secret: Secure channel between rich execution enviornment and trusted execution environment,"

[100] JANG, M., SCHWAN, K., BHARDWAJ, K., GAVRILOVSKA, A., and AVASTHI, A., "Personal clouds: Sharing and integrating networked resources to enhance end user experiences," in *INFOCOM, 2014 Proceedings IEEE*, April 2014.

[101] JEONG, S., LEE, K., LEE, S., SON, S., and WON, Y., "I/o stack optimization for smartphones," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, (Berkeley, CA, USA), pp. 309–320, USENIX Association, 2013.

[102] KAREDLA, R., LOVE, J. S., and WHERRY, B. G., "Caching strategies to improve disk system performance," *Computer*, vol. 27, pp. 38–46, Mar. 1994.

[103] KIM, H., AGRAWAL, N., and UNGUREANU, C., "Examining storage performance on mobile devices," in *Proceedings of the 3rd ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds*, MobiHeld '11, (New York, NY, USA), pp. 6:1–6:6, ACM, 2011.

[104] KIM, H., AGRAWAL, N., and UNGUREANU, C., "Revisiting storage for smartphones," *Trans. Storage*, vol. 8, pp. 14:1–14:25, Dec. 2012.

[105] KIM, J.-M. and KIM, J.-S., "Androbench: Benchmarking the storage performance of android-based mobile devices," in *Frontiers in Computer Education* (SAMBATH, S. and ZHU, E., eds.), vol. 133 of *Advances in Intelligent and Soft Computing*, pp. 667–674, Springer Berlin Heidelberg, 2012.

[106] KOUKOUMIDIS, E., LYMBEROPOULOS, D., STRAUSS, K., LIU, J., and BURGER, D., "Pocket cloudlets," *ACM SIGPLAN Notices*, vol. 47, p. 171, June 2012.

[107] KRISHNAPPA, D. K., KHEMMARAT, S., GAO, L., and ZINK, M., "On the feasibility of prefetching and caching for online tv services: A measurement study on hulu," in *Proceedings of the 12th International Conference on Passive and Active Measurement*, PAM'11, (Berlin, Heidelberg), pp. 72–80, Springer-Verlag, 2011.

[108] KWON, Y., LEE, S., YI, H., KWON, D., YANG, S., CHUN, B.-G., HUANG, L., MANIATIS, P., NAIK, M., and PAEK, Y., "Mantis: Automatic performance prediction for smartphone applications," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, (Berkeley, CA, USA), pp. 297–308, USENIX Association, 2013.

[109] LESNIEWSKI-LAAS, C. and KAASHOEK, M. F., "Ssl splitting: Securely serving data from untrusted caches," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, (Berkeley, CA, USA), pp. 13–13, USENIX Association, 2003.

[110] LI, Z., ZHANG, M., ZHU, Z., CHEN, Y., GREENBERG, A., and WANG, Y.-M., "WebProphet: Automating Performance Prediction for Web Services," in *Proceedings of Networked Systems Design and Implementation (NSDI)*, 2010.

[111] LIANG, J., JIANG, J., DUAN, H., LI, K., WAN, T., and WU, J., "When https meets cdn: A case of authentication in delegated service," in *2014 IEEE Symposium on Security and Privacy*, pp. 67–82, 2014.

[112] LIU, H. and EL ZARKI, M., "An adaptive delay and synchronization control scheme for wi-fi based audio/video conferencing," *Wireless Networks*, vol. 12, no. 4, pp. 511–522, 2006.

[113] LYU, J., KIM, Y., KIM, Y., and LEE, I., "A procedure-based dynamic software update," in *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pp. 271–280, July 2001.

[114] MADHAVAPEDDY, A., LEONARD, T., SKJEGSTAD, M., GAZAGNAIRE, T., SHEETS, D., SCOTT, D., MORTIER, R., CHAUDHRY, A., SINGH, B., LUDLAM, J., CROWCROFT, J., and LESLIE, I., "Jitsu: Just-in-time summoning of unikernels," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, (Oakland, CA), pp. 559–573, USENIX Association, May 2015.

[115] MARTIGNONI, L., POOSANKAM, P., ZAHARIA, M., HAN, J., MCCAMANT, S., SONG, D., PAXSON, V., PERRIG, A., SHENKER, S., and STOICA, I., "Cloud Terminal: Secure access to sensitive applications from untrusted systems," pp. 165–182.

[116] MATTHEWS, J. N., HU, W., HAPUARACHCHI, M., DESHANE, T., DIMATOS, D., HAMILTON, G., MCCABE, M., and OWENS, J., "Quantifying the performance isolation properties of virtualization systems," in *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, (New York, NY, USA), ACM, 2007.

[117] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., and PERRIG, A., "Trustvisor: Efficient tcb reduction and attestation," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, (Washington, DC, USA), pp. 143–158, IEEE Computer Society, 2010.

[118] MCCUNE, J. M., PARNO, B. J., PERRIG, A., REITER, M. K., and ISOZAKI, H., "Flicker: An Execution Infrastructure for TCB Minimization," pp. 315–328.

[119] MCKEEN, F., ALEXANDROVICH, I., BERENZON, A., ROZAS, C., SHAFI, H., SHANBHOGUE, V., and SAVAGAONKAR, U., "Innovative instructions and software model for isolated execution." `https://goo.gl/6mp7ww`.

[120] MICROSOFT, "Microsoft azure: Cloud computing platform & services." `http://azure.microsoft.com/en-us/`.

[121] "Microsoft Bot Framework." https://dev.botframework.com.

[122] MÜLLER, C., LEDERER, S., and TIMMERER, C., "An evaluation of dynamic adaptive streaming over http in vehicular environments," in *Proceedings of the 4th Workshop on Mobile Video*, MoVid '12, (New York, NY, USA), pp. 37–42, ACM, 2012.

[123] NAYLOR, D., SCHOMP, K., VARVELLO, M., LEONTIADIS, I., BLACKBURN, J., LÓPEZ, D. R., PAPAGIANNAKI, K., RODRIGUEZ RODRIGUEZ, P., and STEENKISTE, P., "Multi-context tls (mctls): Enabling secure in-network functionality in tls," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, (New York, NY, USA), pp. 199–212, ACM, 2015.

[124] NEAMTIU, I., HICKS, M., STOYLE, G., and ORIOL, M., "Practical dynamic software updating for c," in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, (New York, NY, USA), pp. 72–83, ACM, 2006.

[125] NETRAVALI, R., SIVARAMAN, A., DAS, S., GOYAL, A., WINSTEIN, K., MICKENS, J., and BALAKRISHNAN, H., "Mahimahi: Accurate record-and-replay for http," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, (Berkeley, CA, USA), pp. 417–429, USENIX Association, 2015.

[126] OWUSU, E., GUAJARDO, J., MCCUNE, J., NEWSOME, J., PERRIG, A., and VASUDEVAN, A., "OASIS: On achieving a sanctuary for integrity and secrecy on untrusted platforms," pp. 13–24.

[127] PAMBORIS, A., BÁGUENA, M., WOLF, A. L., MANZONI, P., and PIETZUCH, P., "Demo:: Nomad: An edge cloud platform for hyper-responsive mobile apps," in *Proceedings of the 13th*

Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '15, (New York, NY, USA), pp. 459–459, ACM, 2015.

[128] PARNO, B., "Bootstrapping trust in a "trusted" platform," in *Proceedings of the 3rd Conference on Hot Topics in Security (HotSec)*, pp. 9:1–9:6, 2008.

[129] PATHAK, A., HU, Y. C., and ZHANG, M., "Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof," in *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, (New York, NY, USA), pp. 29–42, ACM, 2012.

[130] PERRIN, T., "Noise protocol specification." `http://goo.gl/i0PnAU`.

[131] PIERRE, G. and STEEN, M. V., "Globule: A platform for self-replicating web documents," in *Proceedings of the 6th International Conference on Protocols for Multimedia Systems*, PROMS 2001, (London, UK, UK), pp. 1–11, Springer-Verlag, 2001.

[132] RAJ, H., ROBINSON, D., TARIQ, T. B., ENGLAND, P., SAROIU, S., and WOLMAN, A., "Credo: Trusted computing for guest vms with a commodity hypervisor," Tech. Rep. MSR-TR-2011-130, Microsoft Research, 2011.

[133] ROMAN, R., LOPEZ, J., and MAMBO, M., "Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges,"

[134] SADEGHI, A.-R. and STÜBLE, C., "Property-based attestation for computing platforms: caring about properties, not mechanisms," in *Proceedings of the 2004 ACM workshop on New security paradigms*, pp. 67–77, 2004.

[135] SALTZER, J. H., REED, D. P., and CLARK, D. D., "End-to-end arguments in system design," *ACM Trans. Comput. Syst.*, vol. 2, pp. 277–288, Nov. 1984.

[136] SAMTELADZE, N. and CHRISTENSEN, K., "Delta: Delta encoding for less traffic for apps," in *Proceedings of the 2012 IEEE 37th Conference on Local Computer Networks (LCN 2012)*, LCN '12, (Washington, DC, USA), pp. 212–215, IEEE Computer Society, 2012.

[137] SATYANARAYANAN, M., "A brief history of cloud offload: A personal journey from odyssey through cyber foraging to cloudlets," *GetMobile: Mobile Comp. and Comm.*, vol. 18, pp. 19–23, Jan. 2015.

[138] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., and RUSSINOVICH, M., "Vc3: Trustworthy data analytics in the cloud using sgx,"

[139] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., and RUSSINOVICH, M., "Vc3: Trustworthy data analytics in the cloud," Tech. Rep. MSR-TR-2014-39, February 2014.

[140] SEN, J., "Security and privacy issues in cloud computing," *Architectures and Protocols for Secure Information Technology Infrastructures*, pp. 1–45, 2013.

[141] SHERRY, J., LAN, C., POPA, R. A., and RATNASAMY, S., "Blindbox: Deep packet inspection over encrypted traffic," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, (New York, NY, USA), pp. 213–226, ACM, 2015.

[142] SHERRY, J., LAN, C., POPA, R. A., and RATNASAMY, S., "Blindbox: Deep packet inspection over encrypted traffic," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, (New York, NY, USA), pp. 213–226, ACM, 2015.

[143] SU, Y.-Y. and FLINN, J., "Slingshot: Deploying stateful services in wireless hotspots," in *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys '05, (New York, NY, USA), pp. 79–92, ACM, 2005.

[144] SUBRAMANIAN, S., HICKS, M., and MCKINLEY, K. S., *Dynamic software updates: a VM-centric approach*, vol. 44. ACM, 2009.

[145] SUO, H., LIU, Z., WAN, J., and ZHOU, K., "Security and privacy in mobile cloud computing," in *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*, pp. 655–659, IEEE, 2013.

[146] TA-MIN, R., LITTY, L., and LIE, D., "Splitting interfaces: Making trust between applications and operating systems configurable,"

[147] TENNENHOUSE, D. L. and WETHERALL, D. J., "Towards an active network architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 37, pp. 81–94, Oct. 2007.

[148] <TIM@DIERKS.ORG>, T. D., "The transport layer security (TLS) protocol version 1.2." `http://tools.ietf.org/html/rfc5246`, 2013.

[149] TRUSTED COMPUTING GROUP, "Tpm main specification level 2 version 1.2, revision 116," March 2011.

[150] WETHERALL, D., CHOFFNES, D., GREENSTEIN, B., HAN, S., HORNYACK, P., JUNG, J., SCHECHTER, S., and WANG, X., "Privacy revelations for web and mobile apps," in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, (Berkeley, CA, USA), pp. 21–21, USENIX Association, 2011.

[151] WILLIS, D. F., DASGUPTA, A., and BANERJEE, S., "Paradrop: A multi-tenant platform for dynamically installed third party services on home gateways," in *Proceedings of the 2014 ACM SIGCOMM Workshop on Distributed Cloud Computing*, DCC '14, (New York, NY, USA), pp. 43–44, ACM, 2014.

[152] YANG, J. and SHIN, K. G., "Using hypervisor to provide data secrecy for user applications on a per-page basis," in *Proceedings of the 4th International conference on Virtual Execution Environments*, pp. 71–80, 2008.

[153] YEE, B., SEHR, D., DARDYK, G., CHEN, B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., and FULLAGAR, N., "Native client: A sandbox for portable, untrusted x86 native code," in *IEEE Symposium on Security and Privacy (Oakland'09)*, (IEEE, 3 Park Avenue, 17th Floor, New York, NY 10016), 2009.

[154] YOON, H., GAVRILOVSKA, A., and SCHWAN, K., "Attribute-based partial geo-replication system," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, IEEE, 2016.

[155] ZHANG, F., CHEN, J., CHEN, H., and ZANG, B., "Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization,"

[156] ZHOU, W., ZHOU, Y., GRACE, M., JIANG, X., and ZOU, S., “Fast, scalable detection of "piggybacked" mobile applications,” in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY ’13, (New York, NY, USA), pp. 185–196, ACM, 2013.

[157] ZHOU, Z., GLIGOR, V. D., NEWSOME, J., and MCCUNE, J. M., “Building verifiable trusted path on commodity x86 computers,”

[158] ZINK, M., SUH, K., GU, Y., and KUROSE, J., “Watch global, cache local: Youtube network traffic at a campus network: measurements and implications,” vol. 6818, pp. 681805–681805–13, 2008.

VITA

Ketan Bhardwaj was born to Mr. Ramavtar Bhardwaj and Mrs. Sushila Devi in Hisar, a city in Haryana India. He graduated with a Bachelor of Engineering in Electronics and Communications Engineering from Delhi College of Engineering in May 2007. After his Bachelors, he worked with ST Microelectronics, Monsoon Multimedia and Agilent Technologies as system software engineer for 5 years. He is a PhD candidate at the College of Computing, Georgia Institute of Technology, Atlanta, USA advised by Prof. Karsten Schwan and Prof. Ada Gavrilovska. His research at Georgia Tech focuses on Edge Computing.