Fast, scalable and secure onloading of edge functions using AirBox

Ketan Bhardwaj, Ming-Wei Shih, Pragya Agarwal, Ada Gavrilovska, Taesoo Kim, Karsten Schwan
College of Computing, Georgia Institute of Technology, Atlanta, GA
{*ketanbj,mingwei.shih, pragya, taesoo*}*@gatech.edu,* {*ada,schwan*}*@cc.gatech.edu*

***Abstract*—This paper argues for the utility of back-end driven onloading to the edge as a way to address bandwidth use and latency challenges for future device-cloud interactions. Supporting such *edge functions* (EFs) requires solutions that can provide (i) fast and scalable EF provisioning and (ii) strong guarantees for the integrity of the EF execution and confidentiality of the state stored at the edge. In response to these goals, we (i) present a detailed design space exploration of the current technologies that can be leveraged in the design of *edge function platforms* (EFPs); (ii) develop a solution to address security concerns of EFs that leverages emerging hardware support for OS agnostic trusted execution environments such as Intel SGX enclaves; and (iii) propose and evaluate AirBox, a platform for fast, scalable and secure onloading of edge functions.**

***Keywords*-Edge Computing, Edge Cloud, Security, Intel SGX**

I. Introduction

Edge computing has been shown to improve user experience by reducing the user perceived access latency to back-end services, to reduce the cost associated with accessing those services indirectly by consolidating the use of the back-haul bandwidth, and to address the resource constraints of mobile devices.

One direction in research suggests the use of *client-driven* cyber-foraging. Examples of this include offloading systems like MAUI [1], Clone-Cloud [2], Comet [3], that are used to overcome resource constraints of mobile devices and to reduce delays in user perceived responses. However, these approaches are effective only when the response delay is compute bound. Other proposals, such as NOMAD [4], take a client driven approach to minimize the user perceived maximum access latencies. However, there still remains a challenge to control and/or predict the variable Internet access latencies.

Another promising direction is *backend-driven* cyber-foraging that *onloads* appropriate benefits (e.g., caching content, accelerating traffic, etc.) or even some backend functionality (e.g., buffering notifications, aggregating redundant traffic, etc.) near end-users, with a goal of minimizing user perceived latency. *Cloudlet or Edge Clouds* is one such backend-driven approach and has been recently demonstrated for different services, including AppFlux [5], 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), or cognitive assistance application based on Cloudlets [6], that show the feasibility of moving functionality between cloudlet servers and the cloud.

We argue that a back-end driven onloading approach is more pragmatic than client driven offloading. 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.

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 [7] demonstrates opportunities to reduce traffic in the last mile by up to 83% via edge cloud approaches.

Accordingly, we term the onloaded services running on the Edge Cloud as *Edge Functions or EF*, and the underlying software stack as *Edge Function Platform or EFP*.

However, for backend services to utilize edge clouds, the EFPs 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 leaked, that valuable content their EFs store is not stolen, and confidential information about their users is not compromised. Since the edge cloud 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.

In summary, before EF onloading can deliver on its promises, an EFP must satisfy the following goals.

978-1-5090-3322-5/16 $31.00 © 2016 IEEE
DOI 10.1109/SEC.2016.15

Fast and scalable just in time provisioning: Wider mobility patterns of end users and resource constraints (storage) on edge clouds obviate the possibility of static provisioning of all possible EFs in any edge cloud [8]. This led us to explore approaches to provision edge functions just in time. Our goal is to identify the system mechanisms that minimize the time required for EF provisioning and that scale with multiple concurrent requests.

Ensuring security and user privacy: 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 cloud, and handling of user traffic over secure channel. 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, elaborate key exchange protocols or verification schemes, are not desired. Another implicit requirement arising out of deployment models of edge cloud, also shown in Table III, is that an EF cannot rely on system software to meet its security and privacy goals.

Low developer constraints: EF development can benefit from using a common set of software patterns corresponding to obvious edge functions such as caching, aggregation, etc. Still, it is critical that EFPs do not put additional constraints on developers in creating new EFs, i.e., EFs must not be based on specific SDKs, APIs or libraries, etc., outside of the standard system libraries such as libC for Linux and OS-provided system call interfaces.

In this paper, we describe the design of a software platform – AirBox – that strives to find the sweet spot in meeting the above goals and to provide support for fast, scalable and secure 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 I, to conclude that OS containers can provide the right mechanisms and layer for EF provisioning. Further, inspired by recent research [9], [10], [11] which pioneered the use of hardware-level security support such as Intel SGX [12], [13] 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 paper 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 (§II-B).
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 [11] (§II-C).
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 (§IV), and illustrate their implementation with a benchmark that represents the generic functionalities expected from EFs (§V).

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 [8], [14].
- 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 [11] platform.

II. Design space exploration

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 [15] and resource isolation [16]. 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

Technology	Provisioning layer	Flavor
Virtual Machines	Hypervisor	Cloudlets [8]
Containers	OS	Docker [19]
Sandbox	Application	Embassies [20]

Table I: enlisting the approaches, we evaluated for their suitability for Edge Function Platform.

specific compiler tool chains (e.g., Google's NaCl [17]), and a hypervisor-based unikernel (e.g., Jitsu [18] 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 I) 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.

A. 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 [21]. 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 [20] 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 [22]. It seems that there may be a learning curve, but with appropriate skills, developers can overcome those hurdles.

Type	Deployment	Configuration
Mini	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
Mico	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 II: Deployment models and capabilities of edge clouds infrastructure including configurations of experimental test bed.

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 [8], 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 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.

B. Provisioning Performance

Concerning provisioning, the use of virtual machines has been proposed to handle just in time dynamic provisioning of ofoading-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 ofoading, 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 speed. We

Requirement	Mechanisms		
	Cloudlet	Docker	Embassies
Provisioning	Full VM image	Image layering	App Image [20], [14]
Invocation	VM boot	Container startup	Run Embassies [23]
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 [9]
User confidentiality*	mcTLS,split TCP	mcTLS,split TCP	mcTLS,split TCP
Developer Constraint	None	OS	libOS linking, porting

Table III: Mechanisms employed in individual solutions. * Not valid for compromised privileged system software.

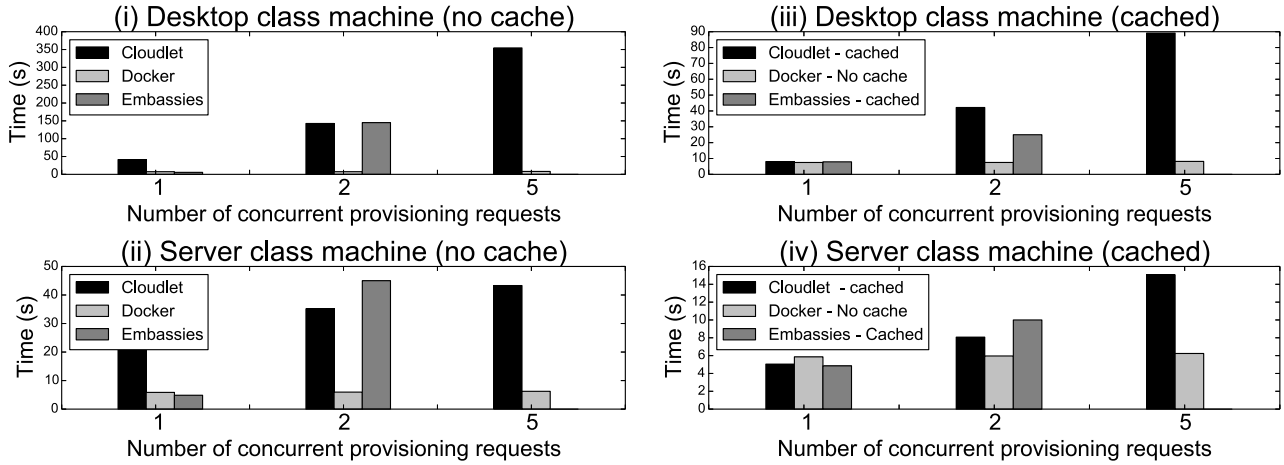

Figure 1: Comparing provisioning time of Docker, Cloudlets, and Embassies without availability of any pre-provisioned base images on (i) desktop and (ii) server class machines and with cached base images on (iii) desktop and (iv) server class machines.

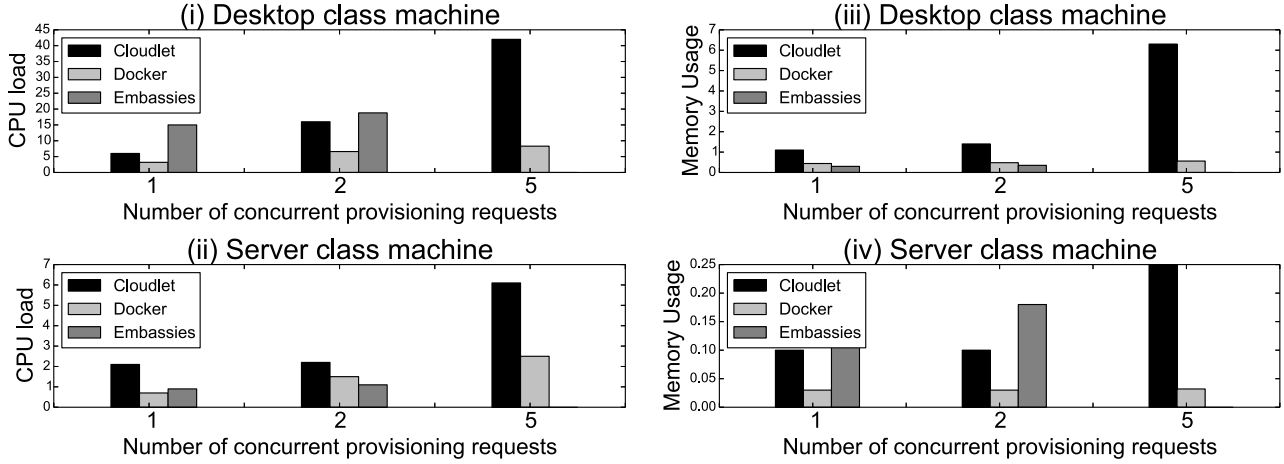

Figure 2: During provisioning of Cloudlets, Docker, and Embassies without availability of base images showing CPU consumption on (i) desktop and (ii) server class machines and memory consumption on (iii) desktop and (iv) server class machines.

present our experimental exploration below.

1) Experimental Setup and Workload:: Table II 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.

2) Experimental Results:: We present the experimental results as part of our design space exploration below:

Scalable provisioning: Figure 1 (i) and (ii) 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 1 (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 [8] 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 1 (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 [14]. 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 [24], but lacks the security guarantees of applications provisioned in Embassies. Earlier measurements have shown that docker containers can boot faster than VMs [15] 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 2(i)) and memory usage (Figure 2(ii)). 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.

C. Security and Privacy

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. [25], 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 II). 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 [25] 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 [26] showed that using a formally verified VMM layer, which results in a smaller trusted code base, can be used for securing execution. InkTag [27] 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 [20], 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 [22] 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 [28], verification via redundant execution [29], trust relationships [30], 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 [9] 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 [10] 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.

D. Summary

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.

III. Background

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 [19].

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 [12], [13].

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

IV. Design

The lack of real world edge cloud deployments severely limits our ability to reason about edge functions, without being affected by the idiosyncrasies of a particular application being posed as an edge function. Further, it is important to note that if there are no latency or bandwidth constraints, all EF services can be provided as the backend services running in the remote clouds.

Before proceeding to design an edge function platform, we want to clarify what we consider as an edge function.

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

Typically, EF is implemented above the network layer, or layer 4, because it requires or uses the backend service's application specific logic to provide benefits in terms of reduced latency, bandwidth consolidation, or both. In AirBox, we focus on two essential elements of EFs, namely *secure provisioning* and *secure EF anatomy*.

A. 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 [31], 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 [25] 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.

B. AirBox 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 [25] (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 3(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

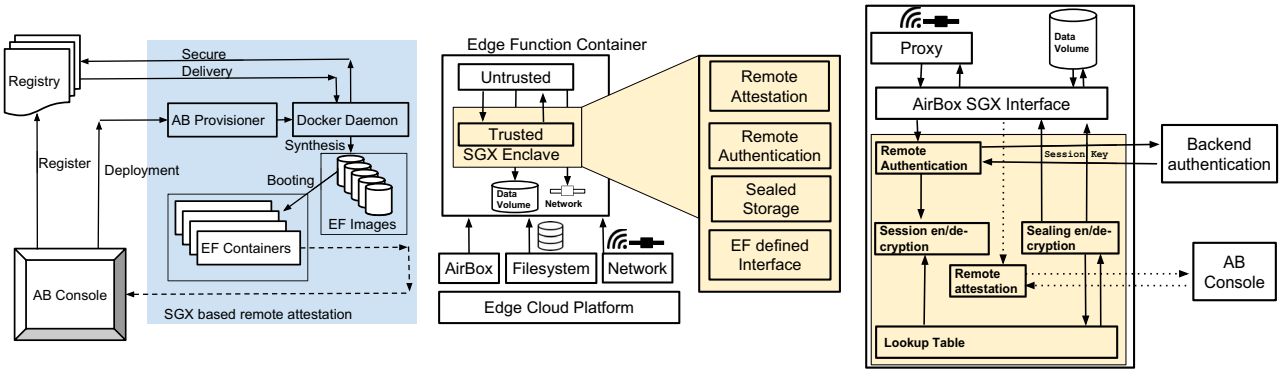

Figure 3: (i) Showing design of AirBox EF provisioning; (ii) Showing overview of secure EF anatomy highlighting the AirBox Open SGX interface; and (iii) implementation of a caching EF.

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.

C. 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:

D. 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 [32] 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 [33] 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.

E. 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.

V. EF 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 typical EFs – the ABC EF benchmark. The implementation of ABC edge functions is based on a simple HTTPS proxy that carries out the following generic operations on web traffic.

Aggregation: An aggregating EF stores multiple requests and/or data from clients and sends a single request to a backend while removing redundant information from the requests. The aggregation function can be supplied by a backend service or by an end user. For instance, an IoT hub on edge cloud could aggregate traffic from different sensors over time while periodically sending information to backend service, resulting in a reduction in bandwidth usage.

Buffering: A buffering EF stores responses from a backend service using edge cloud platform's storage on behalf of one or more client to pre-fetch, buffers it and delivers the response in appropriate context. The context can be based on connecting to a particular edge cloud, e.g., when at home, office etc. or as feature of the backend service to deliver digests of push notifications as opposed individual notifications. This can reduce overheads on a battery operated end client device due to many push notifications and can improve delivery performance for the push notification service. For instance, an end user might authorize a Facebook EF on

a home edge cloud to fetch and buffer his notification and deliver them when he is back home, or an app can use a buffering EF to reduce the push notifications overhead on backend servers for Chrome and native apps which would have to keep polling for the appropriate end users' device to fetch those push notifications.

Caching: A caching EF stores the responses for client request and then, uses those to service other clients or same client with same requests. The caching policy can be service defined, as done in App Sachets [34], which proposes novel caching policies tailored for app installs and updates.

These generic EFs can also be useful as basis for future research in the edge cloud ecosystem by reducing the burden of implementing elaborate use-cases. We describe the implementation of a secure edge function using the caching EF as an example, as shown in Figure 3(iii). 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.

VI. 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 §II-B 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 [35], [34], [36] 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?

A. Provisioning

Figure 4 (i) and (ii) 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 4 (iii) shows how the time spent in attestation requests varies with number of simultaneous requests.

B. 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 5 (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 5 (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. The figure 6 (i) shows the CPU

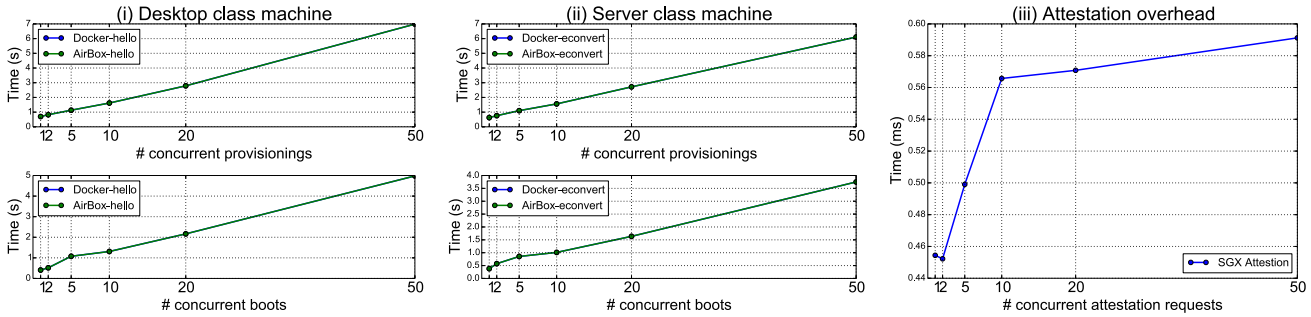

Figure 4: Comparing provisioning time using Docker and AirBox including SGX attestation for a hello world and an image inverting application vs. increasing number of simultaneous provisioning requests using (i) desktop class machine and (ii) server class machine; (iii) time spent in attestation command vs. number of simultaneous attestation requests using windows SDK on SGX hardware.

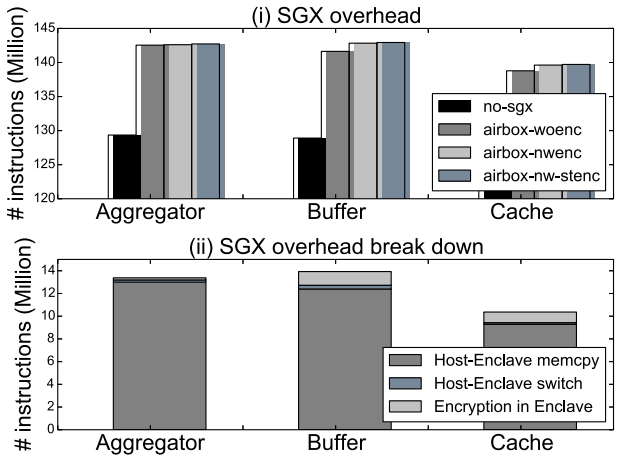

Figure 5: Showing (i) Airbox overhead in ABC use cases due to the use of SGX for security and privacy using OpenSGX; (ii) Showing the break down of the overhead in ABC use cases.

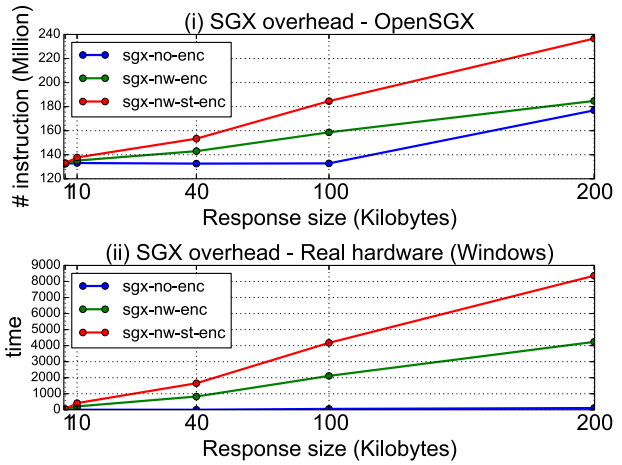

Figure 6: Showing SGX overhead variation with size of data transfer between host and enclave with no encryption, only network interaction encryption and with both network and storage encrypted (i) using OpenSGX and (ii) using Windows SGX SDK.

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 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 6 (ii) 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.

VII. Discussion: 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 [37]. 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. This is important to thwart efforts of spoofing an edge function to access important content (e.g., stealing Netflix videos) combined with the use of the AirBox secure interface to be exposed as a SDK to AirBox EF developers. 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.

VIII. Related Work

Edge Clouds. The emergence of edge cloud infrastructure is evident in research prototypes such as cloudlets [38] and eBoxes [35], and industry initiatives such as Microsoft's micro DCs [39] and ETSI Mobile Edge Computing [37]. Examples of edge cloud nodes include small cells [40], [41], WiFi routers [42], [43], or cloudlet servers [38], 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 Provisioning. Use of virtual machines is proposed in conjunction with VM synthesis [8] 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 [18] 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 [44] 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.

Edge Cloud Security has drawn less attention alongside the fast growth of edge cloud research. Several surveys[45], [46], [47], [48] 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).

IX. Conclusion

In this paper we explore 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 design 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 further provides integrity and security guarantees for edge function computations and the state they use.

Acknowledgement

We would like to thank our shepherd, Padmanabhan Pillai, for his insights during the preparation of the final version of this paper. This work is partially supported through research grants from Intel, VMware, and NSF CNS1148600.

REFERENCES

[1] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Mobisys '10*. ACM.

[2] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Eurosys '11*.

[3] M. S. Gordon, D. A. Jamshidi, S. A. Mahlke, Z. M. Mao, and X. Chen, "Comet: Code offload by migrating execution transparently." in *OSDI '12*.

[4] A. Pamboris, M. Báguena, A. L. Wolf, P. Manzoni, and P. Pietzuch, "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*, ser. MobiSys '15. New York, NY, USA: ACM, 2015, pp. 459–459. [Online]. Available: http://doi.acm.org/10.1145/2742647.2745928

[5] K. Bhardwaj, P. Agarwal, A. Gavrilovska, K. Schwan, and A. Allred, "Appflux: Taming mobile app delivery via streaming," in *TRIOS '15*. Monterey, CA, USA: USENIX Association.

[6] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards wearable cognitive assistance," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '14. New York, NY, USA: ACM, 2014, pp. 68–81. [Online]. Available: http://doi.acm.org/10.1145/2594368.2594383

[7] K. Bhardwaj, P. Agarwal, A. Gavrilovska, and K. Schwan, "Appsachet: Distributed app delivery from the edge cloud," in *MobiCASE 2015*.

[8] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan, "Just-in-time provisioning for cyber foraging," in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '13. New York, NY, USA: ACM, 2013, pp. 153–166. [Online]. Available: http://doi.acm.org/10.1145/2462456.2464451

[9] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 267–283. [Online]. Available: http://dl.acm.org/citation.cfm?id=2685048.2685070

[10] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "Vc3: Trustworthy data analytics in the cloud," Tech. Rep. MSR-TR-2014-39, February 2014. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=210786

[11] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han, "OpenSGX: An Open Platform for SGX Research," in *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2016.

[12] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue, and U. Savagaonkar, "Innovative instructions and software model for isolated execution," https://goo.gl/6mp7ww.

[13] "Intel software guard extensions programming reference," https://goo.gl/sy0tRu.

[14] J. Howell, J. Elson, B. Parno, and J. R. Douceur, "Missive: Fast application launch from an untrusted buffer cache," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 145–156. [Online]. Available: http://dl.acm.org/citation.cfm?id=2643634.2643650

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

[16] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens, "Quantifying the performance isolation properties of virtualization systems," in *Proceedings of the 2007 Workshop on Experimental Computer Science*, ser. ExpCS '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1281700.1281706

[17] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "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. [Online]. Available: http://nativeclient.googlecode.com/svn/data/docs_tarball/nacl/googleclient/native_client/documentation/nacl_paper.pdf

[18] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, J. Crowcroft, and I. Leslie, "Jitsu: Just-in-time summoning of unikernels," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 559–573. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/madhavapeddy

[19] "Docker," https://www.docker.com/.

[20] J. Howell, B. Parno, and J. R. Douceur, "Embassies: Radically refactoring the web," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 529–546. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482676

[21] "Docker on windows," http://goo.gl/oAT5NM.

[22] J. Howell, B. Parno, and J. R. Douceur, "How to run posix apps in a minimal picoprocess," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 321–332. [Online]. Available: http://dl.acm.org/citation.cfm?id=2535461.2535500

[23] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch, "Leveraging legacy code to deploy desktop applications on the web," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 339–354. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855765

[24] "Aufs - layered filesystem @ http://goo.gl/z5l9fc." [Online]. Available: http://goo.gl/Z5l9FC

[25] S. Checkoway and H. Shacham, "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*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 253–264. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451145

[26] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 143–158. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.17

[27] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "Inktag: Secure applications on an untrusted operating system," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 265–278. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451146

[28] S. Garriss, R. Cáceres, S. Berger, R. Sailer, L. van Doorn, and X. Zhang, "Trustworthy and personalized computing on public kiosks," in *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '08. New York, NY, USA: ACM, 2008, pp. 199–210. [Online]. Available: http://doi.acm.org/10.1145/1378600.1378623

[29] Y.-Y. Su and J. Flinn, "Slingshot: Deploying stateful services in wireless hotspots," in *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '05. New York, NY, USA: ACM, 2005, pp. 79–92. [Online]. Available: http://doi.acm.org/10.1145/1067170.1067180

[30] S. Goyal and J. Carter, "A lightweight secure cyber foraging infrastructure for resource-constrained devices," in *Proceedings of the Sixth IEEE Workshop on Mobile Computing Systems and Applications*, ser. WMCSA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 186–195. [Online]. Available: http://dx.doi.org/10.1109/MCSA.2004.2

[31] "Docker trusted registry service," https://goo.gl/lTZWhM.

[32] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, "Blindbox: Deep packet inspection over encrypted traffic," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 213–226. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787502

[33] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D. R. López, K. Papagiannaki, P. Rodriguez Rodriguez, and P. Steenkiste, "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*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 199–212. [Online]. Available: http://doi.acm.org/10.1145/2785956.2787482

[34] K. Bhardwaj, P. Agarwal, A. Gavrilovska, and K. Schwan, "Appsachet: Distributed app delivery from the edge cloud," in *MobiCASE 2015*.

[35] K. Bhardwaj, P. Agarwal, A. Gavrilovska, K. Schwan, and A. Allred, "Appflux: Taming mobile app delivery via app streaming," in *TRIOS 15*.

[36] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards wearable cognitive assistance," in *Mobisys '14*.

[37] "Etsi mobile edge computing," http://goo.gl/Qef61X.

[38] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing '09*.

[39] "Emergence of micro data centers at the edge," http://goo.gl/dVH6ji.

[40] "Qualcomm small cells," http://goo.gl/HEpudP.

[41] "Att small cell deployment plans," http://goo.gl/XdfkfH.

[42] "Att wifi hotspot locations," http://goo.gl/XdfkfH.

[43] "Qualcomm smart gateways," http://goo.gl/BwPc7f.

[44] D. F. Willis, A. Dasgupta, and S. Banerjee, "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*, ser. DCC '14. New York, NY, USA: ACM, 2014, pp. 43–44. [Online]. Available: http://doi.acm.org/10.1145/2627566.2627583

[45] J. Sen, "Security and privacy issues in cloud computing," *Architectures and Protocols for Secure Information Technology Infrastructures*, pp. 1–45, 2013.

[46] R. Illera, S. Ortega, and M. Petychakis, "Security and privacy considerations for cloud-based services and cloudlets," 2013.

[47] H. Suo, Z. Liu, J. Wan, and K. Zhou, "Security and privacy in mobile cloud computing," in *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*. IEEE, 2013, pp. 655–659.

[48] R. Roman, J. Lopez, and M. Mambo, "Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges."