Serverless Computing: A Security Perspective

Eduard Marin[1], Diego Perino[1], and Roberto Di Pietro[2]

[1]Telefónica Research, Spain
[2]Hamad Bin Khalifa University
College of Science and Engineering, Doha-Qatar

Abstract

Serverless Computing is a virtualisation-related paradigm that promises to simplify application management and to solve one of the last architectural challenges in the field: scale down. The implied cost reduction, coupled with a simplified management of underlying applications, are expected to further push the adoption of virtualisation-based solutions, including cloud-computing. However, in this quest for efficiency, security is not ranked among the top priorities, also because of the (misleading) belief that current solutions developed for virtualised environments could be applied to this new paradigm. Unfortunately, this is not the case, due to the highlighted idiosyncratic features of serverless computing.

In this paper, we review the current serverless architectures, abstract their founding principles, and analyse them from the point of view of security. We show the security shortcomings of the analysed serverless architectural paradigms, and point to possible countermeasures. We believe that our contribution, other than being valuable on its own, also paves the way for further research in this domain, a challenging and relevant one for both industry and academia.

keywords: Serverless Computing, Architectures, Security

1 Introduction

Virtualisation technologies have played a crucial role for the wide adoption and success of cloud computing. They allowed cloud providers to simultaneously share their resources with many users by placing their applications (i.e., monoliths) inside virtual machines (VMs), offering strong isolation guarantees while providing users a sense of having an infinite amount of resources. The cited features, together with a pay per use business model that has contributed to lower the TCO, have made cloud computing the most successful computing paradigm of the last decade. However, it came with also some drawbacks: the need for the users to directly manage the VMs is one of those. More recently, new programming models emerged that have drastically changed the way software developers develop and manage applications for the cloud. One of such programming models— *microservices*—relies on decomposing an application into multiple, autonomous, limited scope and loosely coupled components that can communicate with each other via standard APIs. Unfortunately, due to their long startup time and high resource usage, VMs were proven to be inefficient for running such microservices. To tackle the cited limitation, several container technologies (e.g., Docker) were proposed as a lighter alternative. Containers offer increased portability, lower start up time, and greater resource utilisation compared to VMs, simplifying the management of large-scale applications in the cloud. This led cloud providers to adopt container technologies and use them in combination with container orchestration platforms, such as Kubernetes, for fully automating the deployment, scaling, and management of microservice-based applications. However, microservices still require software developers to configure and manage the containers (e.g., libraries and software dependencies) on which they run. Moreover, they rely on a static billing model where users pay a fixed amount according to the allocated resources and not the resources actually consumed. Consequently, microservices are either unsuitable or not viable for certain types of applications.

Serveless (FaaS) paradigm. To tackle the above limitations, a novel paradigm has been conceived: *Function-as-a-Service (FaaS)* [20, 17]. FaaS allows software developers to outsource all infrastructure management and operational tasks to cloud providers, making it possible for them to focus solely on writing the code for their applications [19]—i.e. focusing on the functionalities rather than the infrastructure. FaaS is also widely known as serverless computing—in the following we will mainly use this term. Serverless is the most widely known realisation of FaaS to date (although FaaS could in principle be developed in other ways too). It is worth noting that the term “serverless” does not mean that there are no servers, but rather that software developers do not need to worry about configuring and managing them.

Serverless advantages. With serverless, the application logic is divided into a set of *small* and *stateless* functions, each running within a separate execution environment (e.g., a container) and performing a single task. Functions are typically *short-lived* and are invoked relatively infrequently via both external and internal events (e.g., http requests, table updates, storage modifications or function requests), while storage is provided by separate cloud services shared across users. By doing so, serverless decouples storage from computation, so that they can be provisioned, managed and priced separately. In addition, the cloud provider is now the one responsible for automatically and transparently spawning and managing function instances in worker nodes as well as performing all operational tasks (including security-related ones). These latter tasks include server and OS maintenance, patching, logging, load balancing or auto-scaling. Unlike prior cloud programming models, serverless adopts a pure pay-per-use model where users are only billed based on the resources (e.g., CPU, network or memory) they consume, significantly reducing application deployment cost and helping removing the described cost-inefficiencies of VMs, resulting in an even lower TCO. Besides the advantages serverless provides to software developers in terms of flexibility, scalability, performance and costs, it also offers important benefits to cloud providers. Concretely, as functions are invoked only occasionally and are run for a short period of time, cloud providers can achieve a high degree of co-location in their servers and a more optimal use of their resources that, when carefully planned and orchestrated, can result in an even more profitable model for them.

Serverless players. Cloud providers, such as Amazon [14], Microsoft [6], or Google [13], are already offering serverless computing services to their customers. Meanwhile, the research community has also developed several open-source serverless platforms such as OpenFaaS [12], Knative [11] or Kubeless [2].

Serverless security issues. With the increase in volume and diversity of attacks against the cloud, security and privacy will be a key factor that, if not addressed, could hamper the widespread adoption of serverless computing. At first glance, one could argue that serverless computing is intrinsically more secure than its predecessors due to its characteristics (e.g., the short duration of functions), or due to the fact that it could inherit security features already developed for other virtualisation solutions. However, as we will show in the next sections, serverless brings many new, unique security challenges and it is not immune to security attacks. Implementing serverless applications requires a major change in mindset from software developers, both in the way applications are written as well as in the way they are protected from security attacks. As it will be shown in the following, a new set of security tools is required to help software developers restrict communication to/from functions, to limit functions' permissions to the minimum needed or to protect their data while being stored in cloud services (among others). As of today, serverless security is still a relatively new field, with only a few initial works addressing very specific security issues of the serverless ecosystem.

Contribution. In this paper, we provide several contributions. We first review and categorise state of the art serverless solutions; later, we analyse pros and cons of the introduced architectural categories; further, we assess, from a security perspective, the fundamental principles of the main revised architectural choices. Finally, starting from the highlighted weaknesses, we provide several research directions, appealing to Industry, Academia, and practitioners, to further enhance the security of the serverless ecosystem as a whole.

2 Background and Related Work

In this section we revise the current serverless ecosystem, analysing the 5 main elements any serverless platform is composed of, and then we discuss the currently available security solutions for the introduced ecosystem.

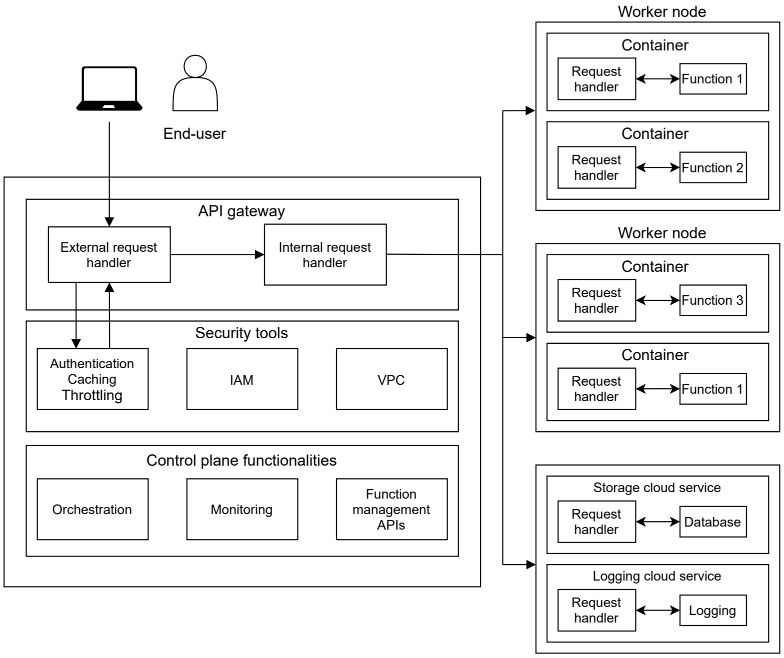

Figure 1: Serverless ecosystem

2.1 Serverless ecosystem

As shown in Figure 1, any serverless platform comprises (at least) the following 5 elements: (i) functions; (ii) API gateways; (iii) (shared) cloud services; (iv) security tools; and, (v) control plane functionalities.

Functions are the elementary component of serverless platforms, providing the elementary logical computational units needed to have the computation progress, and eventually terminate. Software developers can either write the functions themselves, rely on open-source third-party functions (e.g., [5]), or use proprietary functions for which they pay licensing fees. Functions are typically run inside a newly-generated, isolated execution environment within a worker node of the cloud provider's cluster. The type of execution environment being used to run users' functions varies depending on the cloud provider. For example, Amazon executes functions within containers created inside *microVMs*, whereas Google instantiates functions inside sandboxed containers created using *g-Visor*. Functions are executed in response to distinct external and internal types of events specified by application owners such as HTTP requests, modification to objects in storage, table updates, or function transitions. To that end, functions first subscribe to one or more events and then wait for them to occur; functions can learn when one of such events happens via a common bus. Note that only a subset of functions can typically communicate with the outside world.

API gateways are another important component of serverless platforms. They provide a central management service which exposes REST API endpoints to end-users and act as a bridge between end-users and functions. Thus, all communications between end-users and functions have to first traverse API gateways. Normally, API gateways support various security mechanisms for throttling, caching, authenticating and authorising incoming API calls. The latter can be done in various ways (e.g., using identity providers or the IP addresses from which the requests originated).

Shared cloud services. Serverless platforms integrate a wide range of cloud services available to application owners which can be used to extend the functions' functionalities, e.g., to collect various types of data (e.g., Kinesis), achieve long- and short-term storage (e.g., S3 or DynamoDB), manage cryptographic keys, chain several functions with each other (e.g., Step functions) or monitor and log functions' information (e.g., CloudWatch).

Security tools for software developers. In addition to what above described, cloud providers also make available to software developers a set of tools to help them manage the security of their functions. Prominent examples are the Identity and Access Management (IAM) and the Virtual Private Cloud (VPC). The former allows configuring fine-grained access controls to ensure that functions securely access other functions, cloud services and their resources, whereas the latter enables the creation of private, isolated networks for secure communications between functions that belong to the same application.

Control plane functionalities. Serverless platforms comprise a variety of control plane functionalities (not available to application owners). In particular, the control plane includes a set of core services for cloud providers to operate, control and monitor their infrastructures. For example, an orchestrator component handles the process of assigning functions to worker nodes. A monitoring component is also used to periodically check the status of worker nodes and their execution environments. This way, if a failure is detected, the affected functions can be quickly instantiated in other worker nodes. In addition, there is a set of APIs for function management (e.g, to create or update them).

To sum up, serverless platforms are complex and dynamic ecosystems with many distinct components. Therefore, to design a secure serverless ecosystem, one must consider the security provided (or the vulnerability introduced) by each of its components.

2.2 Existing infrastructure-level security controls

Current serverless platforms typically rely on a combination of proprietary and open-source (built into the Linux kernel) security mechanisms to protect functions and the underlying infrastructure against attacks. We cluster the currently provided security mechanisms into 5 distinct categories depending on whether they are used for host hardening, isolation, storage, access control or network purposes. Below, we elaborate on the security mechanisms employed in each of these categories.

Host hardening. As we will show in the next sections, serverless functions can be executed inside various types of execution environments [16], each with its own advantages and disadvantages. One of the main differences among them is the number of functionalities handled by the host OS kernel and the guest kernel, respectively. The execution environments currently used by cloud providers are middle-ground solutions between traditional VMs and containers. Thus, in all of them, functions have access to some parts of the host OS kernel, which increases the attack surface of serverless platforms. By sending maliciously-crafted syscalls to a host OS kernel containing a vulnerability, adversaries could trigger memory leaks, compromise other functions or, even worse, gain elevated privileges in the host. Several Linux kernel features and modules, such as *SELinux*, *apparmor*, *seccomp* and *seccomp-bpf*, are commonly used to mitigate these attacks. All these security mechanisms have in common that they allow cloud providers to define and enforce a security policy that specifies the allowed or disallowed behaviour for a function in terms of system calls, their arguments, and host resources accessed. As a result, these security mechanisms contribute to considerably reducing the host OS kernel attack surface whenever a function is compromised. If this occurs, the adversary will only be able to perform a limited set of actions.

Isolation. To achieve isolation among functions running on the same host, each function is run inside a separate execution environment with its own dedicated *namespace*. Through the use of namespaces, all processes running within an execution environment have their own filesystem, network, processes, interprocess communication and devices. This prevents processes residing in a given execution environment from listing or tampering with processes in other execution environments, making it harder for adversaries to launch attacks. Moreover, *cgroups* are used to control the amount of resources (e.g., CPU, memory or disk I/O) each execution environment can use. This measure ensures that each execution environment gets a fair share of the available resources, thwarting attacks where adversaries aim to over-utilise resources to leave other execution environments without sufficient resources.

Data at rest (Storage). Every time software developers upload a new function to a serverless platform, the cloud provider encrypts the function's code using standard cryptographic algorithms (e.g., AES-GCM [3]). This is done to protect the function's code from unauthorised access while at rest. Similarly, any user data stored or handled by cloud services is kept encrypted while at rest as well. In both cases, the cryptographic keys used by the encryption algorithms can be stored and managed securely using specific cloud services (e.g., AWS Key Management Service). These cloud services often utilise hardware security modules (or similar technologies) to protect cryptographic keys, offering very strong security guarantees to customers.

Network. Whenever the cloud provider instantiates a function in an execution environment of a worker node, the function's code is securely transported to it using a standard cryptographic protocol (e.g., TLS). Moreover, cloud providers have security mechanisms in place such that it is impossible for external users to initiate inbound/outbound network communications with a function without passing by the API gateway first. Additionally, to protect functions from internal threats, application owners can generate virtual, isolated networks, where they can control how their functions communicate with each other and with the outside world. This is achieved using iptables and routing tables. Finally, all communications between functions and between functions and cloud services are protected in transit using secure standard cryptographic protocols like TLS.

Access control. Normally, cloud providers offer various mechanisms for application owners to authenticate and authorise external API calls before passing the requests to the corresponding functions. While this alleviates some attacks by external adversaries, it is not sufficient to prevent abuses by malicious functions. Therefore, cloud providers rely on IAM to let application owners restrict functions' privileges as much as possible. Ideally, functions should always run with least privileges. This way, if a function is compromised, adversaries can only access a relatively small set of cloud services, resources and functions, thus minimising the damage adversaries can cause.

3 Threat model

Scenario. We consider a simple yet realistic scenario that comprises four main actors: (i) cloud providers; (ii) application owners; (iii) application developers; and, (iv) end-users. Let us assume a cloud provider that has a public cloud offering serverless services. The goal of application owners is to leverage serverless computing to easily deploy and manage their applications in the cloud without incurring high costs. Prominent examples of such application owners are Netflix [1] and Realtor [9]. Serverless applications—implemented by application developers—consist of a number of short-lived, stateless functions, each one performing a single task and running inside a fresh, isolated execution environment. However, it is worth noticing that, in practice, cloud providers often opt for reusing execution environments to run multiple instances of the same function to provide better application performance. Finally, we denote as end-users those stakeholders who make use of the application and trigger functions via various types of external events (e.g., a website request).

Adversary Models. In the above scenario, we mainly distinguish between two types of adversaries: (i) *external*; and, (ii) *internal.* External adversaries (i.e., malicious end-users) carry out their attacks from outside the cloud by leveraging any of the existing external APIs serverless platforms offer. These types of attacks are one of the main threats against serverless platforms today. This is because serverless platforms support an ever-increasing number of events to trigger functions, thus giving adversaries many opportunities to inject malicious traffic into functions. These attacks can enable adversaries to retrieve sensitive data or to tamper with the execution of any function or cloud service that receives maliciously-crafted payloads and does not apply proper input sanitation techniques.

On the contrary, internal adversaries refer to those who conduct their attacks from inside the cloud. We distinguish between two types of internal adversaries: (i) adversaries who control one (or several) compromised functions; and, (ii) cloud providers themselves. The former type of adversary can achieve their objectives either by deploying a malicious function or alternatively by compromising a legitimate function in the cloud (e.g., by exploiting a weakness in a third-party library). Afterwards, they can attempt to: (i) escalate privileges to gain full control of other functions or worker nodes; (ii) retrieve or tamper with sensitive data (e.g., session tokens stored in environment tables, data stored in storage services or cryptographic keys kept in cloud services); (iii) gather knowledge about runtime and infrastructure; (iv) upload malicious code to the function; or, (v) conduct various types of Denial-of-Service (DoS) attacks. Finally, we model cloud providers as *honest-but-curious* entities, meaning that they run applications as intended but, at the same time, they try to learn as much information as possible about the ongoing computations.

In next sections we analyse the impact of serverless computing on security, discussing the pros and cons of the paradigm in relationship with its contribution to the security posture of the supported ecosystem.

4 Serverless as a Security Enabler

In this section, we discuss some principles and use cases related to the inherent advantages of serverless from a security point of view.

Functions are stateless and short-lived. The fact that serverless functions are stateless, short-lived, and are executed within ephemeral execution environments, significantly raises the bar for adversaries to successfully execute their attacks. Indeed, while attacks are still possible, serverless imposes strict limits on the time available to the adversary to retrieve sensitive data from functions or to move laterally to perform more sophisticated attacks, similarly to the Intrusion-Resilience Via the Bounded-Storage Model [18].

Less security responsibilities for software developers. In contrast to prior cloud programming models where software developers play an important role in guaranteeing the security of their applications, with serverless most security-oriented tasks are handled by the cloud providers themselves. In general, in serverless one can distinguish between "security of the cloud" and "security in the cloud". "Security of the cloud" is the responsibility of cloud providers and encompasses all measures in place to keep the underlying infrastructure and cloud services (e.g., the execution environments on which functions run or the virtualisation layer) secure from adversaries. This is one of the main advantages of serverless compared to its predecessors regarding security. Instead, "security in the cloud" still assigns the responsibility of security to software developers and mainly refers to the security defences implemented to protect the code written by software developers from adversaries (i.e., application-level security). This latter point includes the security of their code[1], the storage and the accessibility of sensitive data, and the specification of IAM policies. Therefore, serverless security is a shared responsibility between software developers and the cloud provider. The cited concepts are critical ones, and need to be fully seized by software developers, otherwise this could lead these latter ones to ignore security in their applications or to make unrealistic assumptions about the security measures put in place by cloud providers. Despite security still being a shared responsibility between cloud providers and software developers, delegating all infrastructure-related security tasks to cloud providers is considered an effective mechanism to eliminate a wide number of attacks against serverless computing.

Resistance to traditional Denial-of-Service attacks. Due to the elasticity offered by serverless computing, serverless platforms can resist (to a large extent) against DoS attacks aiming to disrupt or crash the application by overloading the servers. Nevertheless, such DoS attacks can lead to a new, serverless-specific attack that leverages the fact that application owners are billed based on the amount of resources their functions consume. By sending many external requests to functions, adversaries can now perform so called Denial-of-Wallet (DoW) attacks with the purpose of significantly increasing the costs for application owners. While some mitigating countermeasures exist against the DoW attacks (e.g., creating a billing alert to notify application owners if they exceed a predefined spending limit), these attacks are not easy to defend against and require additional control measures to detect abnormal behaviour.

5 Serverless as a Security Risk

In this section, we detail several aspects of serverless that can negatively affect security, introducing a few novel vulnerabilities. In the next section, we detail how adversaries can take advantage of the weaknesses explained in this section to carry out attacks. For a security comparison between serverless and its predecessors, we refer the reader to Table 1.

Table 1: Security comparison between monolith applications, microservices, and serverless.

	Monoliths	**Microservices**	**Serverless**
Feasibility of long-lasting attacks	High	High	Low
Main responsible for security	Mostly app owners	Mostly app owners	Shared responsibility
Entry points for adversaries	Few	Few	Many
Resistance to Denial-of-Service attacks	Low	Medium	High
Denial-of-Wallet attacks	Not possible	Not possible	Possible
Communication with other components	None	Medium	High
Visibility of underlying infrastructure	High	High	Low

---

[1] If software developers do not adhere to standard secure coding practices and write their functions' code in an insecure manner, their functions could contain vulnerabilities that can lead to a broad range of attacks.

Larger attack surface Serverless computing offers a significantly larger compared to its predecessors. This is for three main reasons. First, as functions are stateless and are only intended to perform a single task, they are required to constantly interact with other functions and (shared) cloud services to realise their functionalities. However, the definition and enforcement of security policies specifying which functions and cloud services can be accessed by each function in such dynamic and complex environments is very challenging. Second, functions can be triggered by many external and internal event sources (e.g., 47 events in Amazon Lambda [4]) with multiple formats and encoding. This clearly shows that there are many opportunities for adversaries to gain control of functions. Third, serverless platforms include a number of new components and cloud services, many of which are shared across users. Adversaries could leverage such shared components to carry out new forms of side or covert channels that result in attacks aimed to retrieve sensitive data or allow malicious functions to communicate with each other without the cloud provider noticing it. To sum up, serverless computing introduces new, unique security and privacy challenges and gives more possibilities for adversaries to exploit vulnerabilities within functions.

Proprietary cloud provider infrastructures Cloud providers are now the ones responsible for conducting all operational and infrastructure tasks, including those aimed to protect their infrastructures and the hosted applications from internal and external threats. Unfortunately, cloud providers typically keep most information about their infrastructures confidential, making it difficult for security experts to scrutinise the security and privacy of such proprietary serverless platforms. Within the security community, this is commonly known as security-through-obscurity and it is widely known to be a dangerous approach, which may conceal insecure designs' [15]. In response to this, researchers have devoted significant efforts into reverse-engineering and documenting how the serverless platforms of the main cloud providers were designed. This was achieved by performing a wide range of measurements under different settings within their backends using specially-crafted functions. These works showed that the usage of measurement functions can reveal many insights about the way cloud provider backends operate. Yet, there are still many components within serverless platforms that remain unexplored to date whose security level is unknown.

Security vs. performance vs. cost Ideally, cloud providers would like to develop serverless platforms that jointly maximise the security and performance of both their infrastructures and their customers' applications, while keeping their costs as low as possible. However, experience has shown that cloud providers often sacrifice some security to be able to accommodate more users in their infrastructures, to make use of their resources more efficiently or to provide greater performance to their customers' applications. Below, we elaborate more on this with some examples.

Table 2: Comparison of execution environments features. Note that by "containers" and "VMs" we mean cold containers and traditional VMs, respectively. g-Visor refers to Google's sandboxed containers while microVMs corresponds to Amazon's VMs created by Firecracker.

	VMs	**Containers**	**g-Visor**	**microVMs**
Involvement of host OS kernel	Almost none	Almost all	Less than in containers	More than in VMs
App startup times	Very high	Medium	Medium	High
Isolation guarantees	High	Low	Medium	Medium-high
Complexity	High	Medium-low	Medium-low	Medium-high
Written in safe prog. languages	No	No	Yes (Golang)	Yes (Rust)

- **Cold containers vs. warm containers.** When serverless was first introduced, it was proposed to create a new, fresh container (or similar execution environment) isolated from others every time a function is invoked. This approach is suitable in terms of security, however, repeatedly booting a function from scratch inside a newly-generated container (i.e., a cold container) can be an expensive operation latency-wise. This is particularly problematic for serverless because most functions are only run for a very short period of time; consequently the container's booting latency is often similar to the function's execution time. This implies that cold containers can make serverless computing unsuitable for applications with stringent latency requirements. Another reason why the use of cold containers is an issue is that customers are not billed for the time it takes for their containers to boot. To overcome the limitations posed by cold containers, cloud providers have opted for using so-called warm containers, i.e., containers that are reused to run multiple instances of the same function. Warm containers not only reduce the functions' startup times but also improve efficiency, e.g., by keeping and reusing local caches or maintaining long-lived connections between invocations. However, the advantages warm containers offer come at the cost of providing less security guarantees. While warm containers restore the default values in the filesystem within the execution environment every time the function is completed, they contain a small writable /tmp/ disk space to share state across different function invocations. Thus, adversaries who compromise a function, could leverage the fact that the data in /tmp/ is kept across all invocations to continuously execute long-lasting attacks without raising any alarm. To avoid such attacks, application owners can opt for disabling the possibility of reusing the same execution environment to run the same function multiple times. This is particularly useful for those execution environments running functions that perform security-sensitive tasks. Yet, disabling warm containers may not always be a viable option since this can introduce a significant penalty in terms of application performance.

- **Execution environments.** The selection of the execution environment where to run the customers' functions is key for cloud providers since this choice strongly influences the security and performance offered by the serverless platforms. There exist several mechanisms to isolate different applications belonging to different users, each with its own advantages and disadvantages. Unfortunately, it has been demonstrated that the design choices made to improve the security of such execution environments can often negatively affect the system's performance (and vice-versa). For example, VMs offer strong isolation guarantees, but they incur a significant overhead[2]. On the contrary, containers entail less overhead and provide greater resource utilisation at the cost of offering weaker isolation guarantees—all containers in a host share the same underlying host OS kernel. To address the limitations of the aforementioned approaches, it has been proposed to combine VMs and containers together (i.e., by placing all containers of a user in a VM) to benefit from the isolation guarantees of VMs while maintaining (to some extent) the performance advantages offered by containers. However, none of these approaches can jointly satisfy the strict requirements of cloud providers in terms of security and performance.

In practice, cloud providers have opted for developing their own execution environments and open-sourcing their code. For example, Amazon designed Firecracker [7], a solution that builds upon the KVM hypervisor to create and manage VMs—which they call microVMs. Firecracker implements a new virtual machine monitor as well as an API for managing and configuring MicroVMs. Typically, each MicroVM hosts several containers belonging to the same user. Firecracker—used in Lambda since 2018—has been designed with simplicity and minimalism as its key goals, and attempts to reuse existing components wherever possible. With Firecracker, Amazon can not only span new VMs quickly but also securely run thousands of VMs on each host with minimal overhead. Alternatively, Google has developed a user-space application kernel called g-Visor [10]. With g-Visor, each container is given its own application (guest) kernel, which intercepts and handles most of the syscalls invoked by containers itself. Currently, g-Visor supports 237 syscalls out of the over 350 syscalls the Linux Kernel contains. Remarkably, g-Visor only needs to perform 53 syscalls to cover the 237 syscalls, significantly narrowing the interface with the host OS kernel and hence reducing its attack surface considerably. Furthermore, it provides slightly lower application startup times compared to containers [8]. Unfortunately, despite their important advantages and novel designs, these solutions are still rather immature from a security point of view. Hence, research should focus on highlighting their weaknesses and limitations.

Deterministic vs. random scheduling algorithms. Consider the process through which cloud providers assign functions to worker nodes. This can be done using deterministic or randomised scheduling algorithms. From a security point of view, randomised scheduling algorithms are pre-

---

[2] Note that the use of VMs programming language, such as the Java Virtual Machine, are also known to pose a number of security issues.

ferred over deterministic ones because they offer stronger protection against attacks that exploit co-residency with the victim. However, randomised scheduling algorithms do not consider functional aspects such as worker nodes' resource utilisation or function-to-function communication overhead when choosing the worker nodes on which the functions will run. This leads to a non-optimal allocation of functions that can negatively affect the performance of the applications and the underlying serverless infrastructure. To prevent the latter issue, cloud providers typically opt for deterministic scheduling algorithms that lead to a more optimal use of the available resources and less communication overhead. Nevertheless, this approach can be vulnerable to attacks by adversaries who can obtain information about (or tamper with) the way the scheduling algorithms work.

6 Conclusions

In this paper, we have shown that serverless computing on the one hand provides additional security features, while on the other hand it introduces unique security threats and challenges—clearly differentiating itself from current virtualisation technologies, calling for its specific security research thread. In particular, we have reviewed current serverless architectures, categorised the current security threats, and shown actionable security research directions for making serverless the paradigm of choice when looking for virtualisation solutions where security is at premium. We believe that our findings and highlights, other than being interesting on their own, can also pave the way for further research in the field.

References

[1] Netflix & AWS Lambda Case Study. `https://aws.amazon.com/es/solutions/case-studies/netflix-and-aws-lambda/`, 2021.

[2] The Kubernetes Native Serverless Framework. `https://kubeless.io/`, 2021.

[3] AES-GCM authenticated encryption. `https://www.cryptosys.net/pki/manpki/pki_aesgcmauthencryption.html`, 2021.

[4] AWS event types. `https://docs.aws.amazon.com/AmazonCloudWatch/latest/events/EventTypes.html`, 2021.

[5] AWS Serverless Application Repository. `https://aws.amazon.com/en/serverless/serverlessrepo/`, 2021.

[6] Azure Serverless — Microsoft Azure. `https://azure.microsoft.com/solutions/serverless/`, 2021.

[7] Firecracker: Secure and fast microVMs for serverless computing. `https://firecracker-microvm.github.io/`, 2021.

[8] G-Visor's performance Guide. `https://gvisor.dev/docs/architecture_guide/performance`, 2021.

[9] Going Serverless on Images. `https://techblog.realtor.com/going-serverless-on-images/`, 2021.

[10] gVisor. `https://gvisor.dev/`, 2021.

[11] Knative. `https://knative.dev/`, 2021.

[12] OpenFaaS: Serverless Functions, Made Simple. `https://www.openfaas.com/`, 2021.

[13] Serverless Computing Solutions — Google Cloud. `https://cloud.google.com/serverless`, 2021.

[14] What is AWS Lambda? `https://docs.aws.amazon.com/lambda/latest/dg/welcome.html`, 2021.

[15] Anderson, R. J. *Security Engineering: A Guide to Building Dependable Distributed Systems*, 2 ed. Wiley Publishing, 2008.

[16] Anjali, Caraza-Harter, T., and Swift, M. M. Blending Containers and Virtual Machines: A Study of Firecracker and GVisor. In *Proceedings of ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (2020), pp. 101–113.

[17] Castro, P., Ishakian, V., Muthusamy, V., and Slominski, A. The Rise of Serverless Computing. *Communications ACM 62*, 12 (2019), 44–54.

[18] Dziembowski, S. Intrusion-resilience via the bounded-storage model. In *Theory of Cryptography* (Berlin, Heidelberg, 2006), S. Halevi and T. Rabin, Eds., Springer Berlin Heidelberg, pp. 207–224.

[19] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N. J., Gonzalez, J. E., Popa, R. A., Stoica, I., and Patterson, D. A. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* (2019).

[20] Schleier-Smith, J., Sreekanti, V., Khandelwal, A., Carreira, J., Yadwadkar, N. J., Popa, R. A., Gonzalez, J. E., Stoica, I., and Patterson, D. A. What Serverless Computing is and Should Become: The next Phase of Cloud Computing. *Communications ACM 64*, 5 (2021), 76–84.