System Call Interception for Serverless Isolation

Nishant Somy
IIT Kharagpur, India
somy1997@gmail.com

Abhijit Mondal
IIT Kharagpur, India
abhijit.manpur@gmail.com

Bishakh Ghosh
IIT Kharagpur, India
ghoshbishakh@gmail.com

Sandip Chakraborty
IIT Kharagpur, India
sandipc@cse.iitkgp.ac.in

CCS CONCEPTS

• **Networks** → **Cloud computing**; • **Software and its engineering** → *Process management*.

KEYWORDS

Serverless computing, function isolation, zero cold-start

ACM Reference Format:
Nishant Somy, Abhijit Mondal, Bishakh Ghosh, and Sandip Chakraborty. 2020. System Call Interception for Serverless Isolation. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20 Demos and Posters), August 10–14, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3405837.3411391

1 OVERVIEW

Serverless functions [6, 9, 13, 14], like AWS Lambda [3] or Google Cloud Functions [7], are new techniques for running short-lived workloads over a cloud, which are particularly preferred by users for their easy deployment, fine-grained billing and automatic scaling. Unlike traditional cloud offerings such as VMs and containers, these functions are **stateless**, where each function execution starts with a fresh state of memory, disk, and other resources. A multi-tenant serverless cloud platform has two primary components, a **gateway** controlled by the cloud service provider (CSP) and the **user functions** which are users' programs. A typical serverless function execution is **stateless** and involves broadly the following steps: (1) User request is received by the gateway, (2) Gateway executes the function and passes the request arguments, (3) The function performs necessary computation, (4) Optionally, it can call other functions through the gateway or access external Database, and (5) It returns the result to the user.

Thus for supporting multi-tenancy, CSPs are required to ensure isolation for the user functions in three aspects [4, 12] - **(a) function-function isolation:** one user function cannot access the memory, files, or any resource of another function, **(b) function-host isolation:** access to host resources and functionality like system file access, accepting network connections, process execution are not allowed by the user functions. **(c) restriction in resource utilization:** there must be a limit in CPU/memory/disk usage so that one function workload cannot hamper the execution of other functions. Moreover, each function has a limited maximum duration of execution (e.g. 15 minutes for *AWS Lambda*). Existing serverless platforms rely upon VMs and containers-based sandboxing, which were originally designed to support multi-tenancy for traditional web-applications. VMs use hypervisor based virtualization, whereas containers share the host OS kernel and provide isolation by using *namespaces* and *control groups* (`cgroups`). While both these techniques provide the isolation and security guarantees, they come with very high overheads in terms of *cold-start latency* and resource overheads [5]. Existing works [1, 2, 5, 10, 11] primarily try to optimize the VM and containers for serverless operations, through lightweight VMs [1] or containers [2, 5, 10, 11], therefore cannot eliminate the cold-start latency completely.

In contrast, we take into consideration the properties of serverless functions and their limited access to resources to design and implement a novel isolation architecture from ground up. Our proposed sandbox architecture, called NOVA, is designed specifically for serverless workloads, and uses OS level system-call monitoring and whitelisting to eliminate any cold-start latency as well as resource overheads. Unlike programming language specific isolation like *V8 isolates* used in Cloudflare[1], NOVA is language agnostic. Results demonstrate that NOVA sandboxing has zero cold-start latency and negligible I/O overhead.

2 SYSTEM DESIGN

We propose NOVA sandboxing to achieve function isolation with zero cold-start latency and minimal resource & I/O latency overhead by exploiting a fundamental principle of the OS - **system calls**. System calls are the methods to transfer control from the user-space to the kernel-space, and thus the only way for a process to interact with other processes and access the hardware & software resources in the host system. To support isolation, NOVA monitors system calls originated from a function and selectively allows them based on the properties of a serverless function. However, using such technique comes with its own set of challenges. Some system calls are essential to start execution of the function, whereas, some are required by them to load the necessary software libraries and resources. Moreover, not all host processes should be affected, and this isolation should only affect the serverless workloads. Another major problem is passing user input and retrieving the result from these functions while preserving the isolation. To handle these challenges NOVA uses two primary components (Figure 1) - **(a) NOVA Interceptor** - module to isolate the function execution environment from the rest of the system. **(b) NOVA Gateway** - for receiving user requests and executing the appropriate function with proper arguments.

2.1 NOVA Interceptor:

The task of the NOVA Interceptor is to restrict serverless functions' access to host resources by blocking its system calls for isolation,

[1]https://workers.cloudflare.com/ (accessed March 25, 2021)

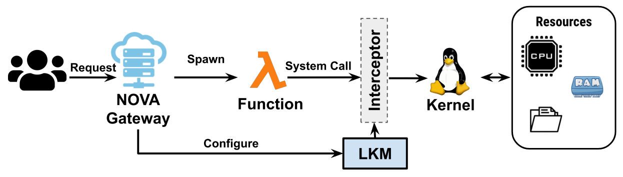

Figure 1: System architecture

while still allowing some calls to preserve its functionality. NOVA Interceptor filters system call from the kernel space instead of the users-space. The *Linux* kernel has a subroutine to handle the software interrupt triggered by the user-space program acting as the single point entry for the system call to the kernel. Although changing this subroutine to support our filtering feature can solve the problem, this will require a kernel upgrade. However, the subroutine uses a **`system call table`** to call the appropriate handler for the system call. Therefore we exploit the `system call table` and modify its entries from a Loadable Kernel Module (LKM) for changing the behaviour of each system call. We implement our own proxy handlers for each call which decides what system calls to allow. NOVA Interceptor should only intercept the calls from the serverless workloads, and not from other host processes. So, the proxy handler selectively processes some calls by matching the parent process id (`ppid`) of the calling process with predefined process id (`pid`) of the NOVA Gateway. If it does not match (for host processes), the proxy handler immediately calls the original system call handler.

The LKM restricts file system access by blocking `open`, `stat`, `fstat`, and `lstat` system calls. For network isolation, we allow only TCP connection in client mode from the serverless function by restricting `socket` system call to stream type socket and block `bind`, `listen`, `accept` and `setsockopt` system calls. This abides by the serverless property in which the functions can only be accessed through the CSP's gateway. NOVA restricts all kind of inter-process communication. The LKM blocks `pipe`, `mkfifo`, `shmget` and other system calls related to IPC. It is also essential to restrict process operation from the function by restricting system calls like `fork`, `exec`, `clone`. However, we allow the `kill` system call to send signals to itself only. Some system calls are dependent on other system calls (like `read`, `write`, etc.), so we do not need to block them explicitly. The complete list of the blocked system call is listed at https://github.com/ghoshbishakh/novaisolation.

2.2 NOVA Gateway:

NOVA uses a Common Gateway Interface (CGI) as the gateway, with some modifications to support **NOVA Interceptor**. CGI suites this particular use-case since the input and output from the serverless functions are passed through environment variables and `stdout` respectively, both of which are not blocked by *NOVA Interceptor*. It is developed in such a way that it first configures the LKM with its own `pid` and isolation configuration. After that, it starts listening for the user requests. Whenever a new request arrives, it executes the corresponding function using CGI and responds the output, as shown in Figure 1. The CGI runs the executable for each function request in a new child process. As the LKM blocks all kinds of open system calls, the function can not load any dynamically loadable libraries. Instead of partially allowing open system call, we avoid all kinds of dynamic linking by generating a statically linked function program only, thus preserving the properties of a serverless function. Although the executable size is a little large, it saves us from solving the problem of opening a shared object from the disk.

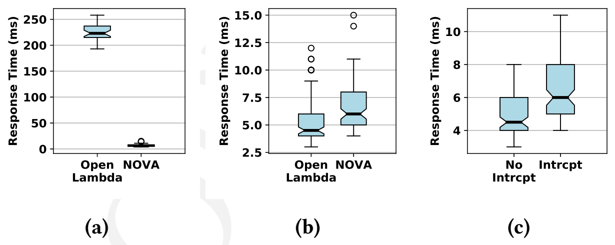

Figure 2: Response time during (a) cold-start and (b) normal operations; (c) Overhead due to system call interception

3 PRELIMINARY RESULTS

To evaluate NOVA, we have implemented it on Linux kernel 4.15 as an LKM, and the performance is compared with OpenLambda [8]. A VM with 2 vCPUs and 3GB memory was emulated as the platform to host the function. The workload was a simple program for finding the n^{th} Fibonacci number, with n given as a user input through the query string of an HTTP request. When the NOVA gateway receives the HTTP request, it prepares the set of environment variables for the function and the `QUERY_STRING` variable. The nova gateway creates two (POSIX) pipe to send and receive request-response data from the function and calls `fork` to create the process. After that, the new process duplicates the file descriptors from the `pipe` to the `stdin` and `stdout` using the `dup2` system call and closes the original file descriptors of the pipe. At this point, the gateway is ready to load the function using `execv`. The LKM does not block any system call before the `execv` call; however, after the `execv` call, the LKM starts monitoring the system calls. As the Fibonacci function is straightforward, it does not require any system call as such. However, to return the results, it needs to call the `write` system call on `stdout`. At the end, the function calls `exit` to terminate the execution. NOVA interceptor does not block `write` and `exit` system calls.

For each experiment, about 100 user requests were used. Figure 2a shows significant reduction in cold-start latency since NOVA effectively removes any cold start phase; for normal operations (Figure 2b) the latency is almost similar. To analyze the reason behind the marginal increase in normal operation latency, we performed the same experiment by disabling the NOVA Interceptor while keeping everything unchanged. Figure 2c indicates that this overhead is entirely from intercepting and processing system calls. In terms of memory usage we found that NOVA sandboxes use only ~6MB, while OpenLambda functions use ~35MB. Thus these initial results suggest that NOVA is lightweight with negligible overhead compared to container-based serverless platforms.

REFERENCES

[1] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX NSDI*. 419–434.

[2] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *USENIX ATC*. 923–935.

[3] Amazon. 2020. AWS Lambda. https://aws.amazon.com/lambda/. (2020). [accessed March 25, 2021].

[4] Ankit Bhardwaj, Meghana Gupta, and Ryan Stutsman. 2020. On the Impact of Isolation Costs on Locality-aware Cloud Scheduling. In *USENIX HotCloud*.

[5] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen. 2020. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *25th ASPLOS*. 467–481.

[6] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *USENIX ATC*. 475–488.

[7] Google. 2020. Cloud Functions. https://cloud.google.com/functions/. (2020). [accessed March 25, 2021].

[8] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. 2016. Serverless computation with OpenLambda. In *USENIX HotCloud*.

[9] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[10] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov. 2019. Agile cold starts for scalable serverless. In *USENIX HotCloud*.

[11] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. 2018. SOCK Rapid task provisioning with serverless-optimized containers. In *USENIX ATC*. 57–70.

[12] Yuxin Ren, Guyue Liu, Vlad Nitu, Wenyuan Shao, Riley Kennedy, Gabriel Parmer, Timothy Wood, and Alain Tchana. 2020. Fine-Grained Isolation for Scalable, Dynamic, Multi-tenant Edge Clouds. In *USENIX ATC*. 927–942.

[13] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 1063–1075.

[14] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. 2018. Peeking behind the curtains of serverless platforms. In *USENIX ATC*. 133–146.