Bridging language- and hardware-based isolation for scalable and secure serverless runtimes

Work-In-Progress (WIP) paper

1 MOTIVATION

The serverless cloud computing model is gaining significant adoption as it enables applications to harness extreme elasticity and fine-grained billing while freeing developers from infrastructure management hurdles [4, 13]. Serverless is often offered in the form of an event-based Function-as-a-Service programming model where applications are split into lightweight and fast-executing logic units called functions which are launched automatically by the serverless platform upon invocation. As a result, serverless sparked a lot of interest among practitioners which led to the development of many serverless applications in image and video processing [6], machine learning [3], data processing [9, 12, 16], and web-based applications [7].

Serverless functions have different performance characteristics compared to serverfull or even to microservices, particularly when it comes to memory footprint and execution time. A recent study [14] showed that most serverless applications run for at most 1 second and use up to 150 MBs of memory. For such fine-grained computations, traditional hardware and system isolation techniques (processes, containers, VMs) are inefficient at protecting each unit of computation from other units.

2 LIMITATIONS OF EXISTING ISOLATION

Despite their strong isolation properties, virtual machines (VMs) and containers are inefficient in the context of isolating fine-grained serverless computations. First, high memory redundancy stems from having multiple copies of the same application and language runtime in memory. Second, high cold start latency since every time a new sandbox needs to be created to host a function invocation, a new virtual machine or container needs to be launched. To mitigate these problems, recent works have proposed using a single language runtime (and therefore a single VM/container) to host multiple function invocations.

Process-based isolation. SAND [2] and SOCK [10] propose a zygote process that forks into a child process used to handle the invocation. This approach only supports Python as it is not straightforward to apply fork to most other commonly-used runtimes without expensive reinitialization mechanisms [2, 5, 10]. For example, most Java and Javascript runtimes contain background threads that deal with Garbage Collection and Just-in-Time compilation. Implementing the fork approach for these runtimes would require properly re-initializing all threads in the child process, which involves complex and error-prone changes to the runtime.

Runtime/compiler-based SFI. Photons [5] propose a bytecode rewriting tool that isolates global state between concurrent function invocations running on the same Java Virtual Machine (JVM). However, this technique is Java-specific and requires manual intervention when Java's reflective features (reflection for example) are utilized by the function code. Faasm [15] relies on Wasm to deploy sandboxes within a single address space but also has limited support for the most popular serverless languages (Python, Javascript, and Java).

MPK-based isolation. Faastlane [8], on the other hand, offers stronger memory isolation through Intel Memory Protection Keys [11] (MPKs) between functions within a single process/runtime. Intel MPKs have recently been introduced as an isolation mechanism within a single address space. MPKs split virtual memory in domains that can be assigned to one or more threads. However, MPKs are limited to 16 memory domains, preventing their wide use to isolate concurrent function invocations (only up to 16 invocations would be handled in a single address space).

3 KEY INSIGHT

To lift the domain count limitation of existing hardware mechanisms and scale isolation to real-world serverless workloads, our approach leverages the following key insight: managed language runtimes already offer memory isolation while the function is running managed code (i.e., code managed by the runtime), hence, non-language-based isolation techniques are only required when there is a control-flow transition to native code (i.e., code not managed by the runtime).

In managed languages, function code can only handle references within the object graph managed by the sandbox and cannot access arbitrary memory positions. Therefore, code executing within the sandbox (we refer to it as managed code) is not able to access memory locations belonging to other concurrent functions. This guarantee is offered by the compilers and runtime, which dynamically inject and execute a number of integrity checks (e.g., null pointer checks, buffer overflow checks). Taking advantage of such language-based guarantees, it is possible to delay the utilization of MPK domains until there is a transition from managed code

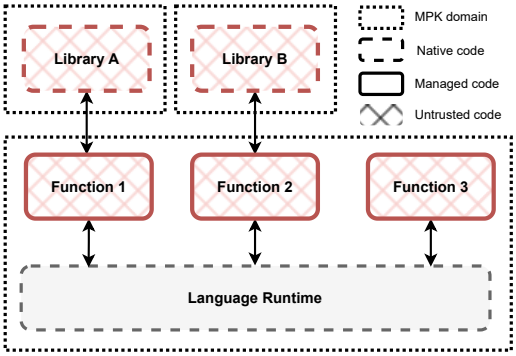

Figure 1: Memory domain distribution when running three concurrent function invocations.

into native code, which runs outside the sandbox, where the guarantees no longer apply. For example, if a Java function calls a native library (through the Java Native Interface), an MPK domain is only used isolate the library data and code while the function is running in the native mode (see Figure 1). Functions that do not rely on external libraries will naturally not use any MPK domain.

4 BRIDGING LANGUAGE AND HARDWARE ISOLATION

This design has several advantages. First, isolation guarantees are offered to all functions but will only be enforced if the function calls into external libraries (native code). Second, even if most functions need native code execution, MPK domains can be shared over time. A single domain can be shared not only across invocations that do not overlap in time but also across invocations that overlap as long as they don't run native code simultaneously (see Figure 2. Third, since only native code will be moved to a different MPK domain, remapping becomes more efficient as the bulk of the memory (managed by the runtime) is not required to be in an MPK domain since the isolation is still guaranteed by the compiler/runtime.

Capturing Managed to Native code calls Memory isolation can be enforced at the managed execution boundaries by analyzing the function code statically and inserting code snippets to intercept transitions from managed code to native code. Those transitions are handled by moving the calling thread and the native code library that is being invoked into a free MPK domain. When the native code call returns, we move the thread and library back to the original MPK domain so that other functions could use it for calling native code. It is also possible to collect metadata information about if and when invocations call native code. Such information could be useful to i) make better decisions at the higher level scheduling to reduce the number of function invocations

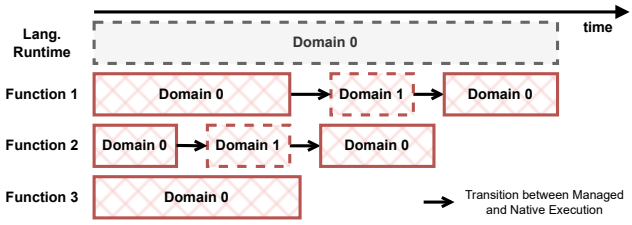

Figure 2: Memory domain distribution along the execution time of three concurrent function invocations.

that rely on external libraries on the same process, and ii) push some of the MPK domain initialization/registration to the background.

We consider the language runtime and compiler as trusted entities and therefore, interactions between the function code and the language runtime do not need to be monitored even if these components are written in non-managed languages such as C/C++. We do not trust user-provided code which (may include a mix of managed and native code) as it can potentially contain bugs or even malicious code.

Implementation and Evaluation We are building a prototype using GraalVM Native Image [17], an ahead-of-time compiler for Java applications. It supports memory isolates [1] which we use to confine the execution of a particular function invocation in a separate memory heap. Furthermore, Native Image limits dynamic class loading, ensuring that all the reachable code is known at compile. Managed to native control-flow transitions are being intercepted to update the calling thread's permissions (i.e., revoking access to the default MPK domain). In addition, we also intercept native code loading and keep track of which pages belong to which library such that when a managed to native transition is issued, we are able to move the pages belonging to the library that is being called into a free MPK domain.

We are looking into a number of opensource serverless benchmarks and analyzing how much time is spent on native code and the overall number of transitions. We intend to evaluate our prototype on a realistic serverless setup deployed on top of a serverless platform such as OpenWhisk and measure the overall cost reduction compared to running all invocations in completely separate MPK domains. Updating the MPK permissions of a particular thread takes less than 25 CPU cycles and updating the domain of a range of memory pages belonging to a native library is very efficient [11]. We envision our design could significantly improve the scalability (while retaining security guarantees) of serverless platforms when there is a mix of functions with different characteristics of frequency and duration of native code utilization.

[1] Memory isolates are also supported by other runtimes such as V8 [1].

REFERENCES

[1] 2023. V8 JavaScript Engine. https://v8.dev/. Accessed: 2023-02-23.

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[3] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A Serverless Framework for End-to-End ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/3357223.3362711

[4] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The Rise of Serverless Computing. *Commun. ACM* 62, 12 (nov 2019), 44–54. https://doi.org/10.1145/3368454

[5] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. 2020. Photons: Lambdas on a Diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 45–59. https://doi.org/10.1145/3419111.3421297

[6] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'17)*. USENIX Association, USA, 363–376.

[7] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 152–166. https://doi.org/10.1145/3445814.3446701

[8] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 805–820. https://www.usenix.org/conference/atc21/presentation/kotni

[9] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 115–130. https://doi.org/10.1145/3318464.3389758

[10] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[11] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 241–254. https://www.usenix.org/conference/atc19/presentation/park-soyeon

[12] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 131–141. https://doi.org/10.1145/3318464.3380609

[13] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What Serverless Computing is and Should Become: The next Phase of Cloud Computing. *Commun. ACM* 64, 5 (apr 2021), 76–84. https://doi.org/10.1145/3406011

[14] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[15] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker

[16] Michal Wawrzoniak, Ingo Müller, Rodrigo Bruno, and Gustavo Alonso. 2021. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *International Conference on Innovative Data Systems Research*.

[17] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize Once, Start Fast: Application Initialization at Build Time. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 184 (oct 2019), 29 pages. https://doi.org/10.1145/3360610