A Secure, Fast, and Resource-Efficient Serverless Platform with Function REWIND

Jaehyun Song and Bumsuk Kim, *Sungkyunkwan University;*
Minwoo Kwak, *Yonsei University;* Byoungyoung Lee, *Seoul National University;*
Euiseong Seo, *Sungkyunkwan University;* Jinkyu Jeong, *Yonsei University*

https://www.usenix.org/conference/atc24/presentation/song

This paper is included in the Proceedings of the
2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

A Secure, Fast, and Resource-Efficient Serverless Platform with Function REWIND

Jaehyun Song[1], Bumsuk Kim[1]*, Minwoo Kwak[2], Byoungyoung Lee[3], Euiseong Seo[1], Jinkyu Jeong[2]
[1]*Sungkyunkwan University,* [2]*Yonsei University,* [3]*Seoul National University*
{jaehyun.song, bumsuk.kim}@csi.skku.edu, {minwoo.kwak, jinkyu}@yonsei.ac.kr,
byoungyoung@snu.ac.kr, euiseong@skku.edu

Abstract

Serverless computing often utilizes the warm container technique to improve response times. However, this method, which allows the reuse of function containers across different function requests of the same type, creates persistent vulnerabilities in memory and file systems. These vulnerabilities can lead to security breaches such as data leaks. Traditional approaches to address these issues often suffer from performance drawbacks and high memory requirements due to the extensive use of user-level snapshots and complex restoration process.

The paper introduces REWIND, an innovative and efficient serverless function execution platform designed to address these security and efficiency concerns. REWIND ensures that after each function request, the container is reset to an initial state free of any sensitive data, including a thorough restoration of the file system to prevent data leakage. It incorporates a kernel-level memory snapshot management system, which significantly lowers memory usage and accelerates the rewind process. Additionally, REWIND optimizes runtime by reusing memory regions and leveraging the temporal locality of function executions, enhancing performance while maintaining strict data isolation between requests. The prototype of REWIND is implemented on OpenWhisk and Linux and evaluated with serverless benchmark workloads. The evaluation results have demonstrated that REWIND provides substantial memory savings while providing high function execution performance. Especially, the low memory usage makes more warm containers kept alive thereby improving the throughput as well as the latency of function executions while providing isolation between function requests.

1 Introduction

Serverless computing, also known as Function-as-a-Service (FaaS), has significantly gained traction in cloud computing, evident from its adoption by major cloud vendors. This shift from traditional virtual machine-based cloud computing or Infrastructure-as-a-Service (IaaS) offloads system operation responsibilities like auto-provisioning, auto-scaling, and load-balancing from the client to the platform itself [26, 36]. This model frees clients from the complexities of managing the entire software stack, from operating systems (OSs) to applications, allowing them to focus more on application or function development. Driven by the benefits of faster development and continuous integration/continuous delivery (CI/CD) processes [31], serverless computing has become increasingly crucial.

The FaaS platforms usually process requests for a serverless function in containers dedicated to that function [39]. This design allows function's processing capability to dynamically scale in or out, responding to changes in request demand. To support this, function containers must be stateless, and function's data should be ephemeral, being maintained only for the duration of a single request execution. This architecture fundamentally provides robust security. The use of multiple containers offers isolated execution environments across requests. Consequently, even if one container is compromised, its impact does not extend to requests being processed in other containers. Additionally, function containers' ephemeral nature further enhances security, as any residual memory or file data within a container disappears quickly.

Containers require hundreds of milliseconds to launch [17]. To remove this startup latency of function containers out of the function execution, serverless platforms typically reuse these containers for subsequent requests of the same function [6, 18, 19, 33, 37, 53, 58]. Unfortunately, this practice of employing *warm* containers may create security vulnerabilities, such as the potential exposure of sensitive data or the risk of malware attacks from previous requests [4, 13].

One of the most straightforward solutions to address these security threats is to restore function container's state to its initial one before processing the next request. The FaaS platform should checkpoint the initial state of the container as a snapshot when it is first launched, and restore function container's state from this snapshot after the execution of

*Currently at Samsung Electronics

each request when reusing the container. By employing this checkpoint-restore (C/R) approach, even when executing requests in warm containers, it ensures a secure execution environment free from the influence of previous requests [4].

Although effective in mitigating security risks associated with reusing containers, the C/R approach can lead to significant execution delays caused by the restoration process. Given the prevalent use of FaaS for lightweight microservices, such delays present a challenge for real-world deployment. Moreover, the need for additional DRAM to store snapshots, proportional to function container's initialized memory size, limits the number of containers a server can host.

In this paper, we propose REWIND, a fast and secure serverless platform that enhances the warm container reuse technique with added isolation between function requests. REWIND goes beyond just targeting memory and includes processes and file systems in its system-wide checkpoint of the initial state. Upon container reuse, REWIND performs a restore operation from the snapshot, which we call *rewind*.

REWIND features a novel *buddy page table* structure in the OS kernel, efficiently maintaining both snapshot page mappings and page mappings altered during function execution. This approach simplifies checkpointing by maintaining current page mappings without duplicating the entire address space. This also drastically reduces the memory requirements for storing the snapshot compared to the C/R approaches. If new pages are allocated or page mappings change during execution, the altered mappings will be recorded alongside the snapshot mappings in the buddy page table. The rewind operation simply eliminates any new or modified mappings from the buddy page table, returning the container to its pristine state for the next request. As a result, rewind completes significantly faster than C/R schemes' restoration operation.

REWIND's memory management structure not only enhances security but also improves the speed of memory operations during function execution. It records all changes in the page table. Since serverless functions repetitively perform the same operations for requests, memory management tasks, including memory population, tend to be repeated identically during execution. REWIND reduces memory management overhead by not releasing but unmapping the pages used for memory allocation and CoW operations from the previous execution, sanitizing, and then reusing them in subsequent executions.

Since REWIND creates a snapshot of process information at the OS level and uses it for the rewind operation, it can eliminate threats from malicious processes as well that might exist due to exploitation like a *rootkit* [22, 57]. Additionally, by exploiting the characteristics of overlay file systems [44] popularly used by containers, REWIND creates a layer for writes that occur during function execution, and later removes the layer as a part of the rewind operation. This can prevent data leakage through the file system with minimal overhead.

We implemented REWIND on the combination of OpenWhisk [42], an open-source serverless computing platform, and the Linux kernel [34]. We evaluated a few benchmark workloads with diverse characteristics [9, 29, 62]. The evaluation results were compared against the state-of-the-art C/R-based solution in terms of performance and memory overhead.

This paper has the following contributions:

- We propose REWIND, an approach to enforce isolation of consecutive function requests handled within a shared function container at a low performance and memory cost. Different from previous works, REWIND provides isolation of memory as well as processes and file systems.
- Under the hood, we propose a buddy page table, REWIND's core page table structure extended to support capturing and restoring a snapshot of a virtual memory while keeping the number of pages and memory copy operations low for managing a snapshot.
- We demonstrate the performance benefit and memory efficiency of REWIND with realistic serverless workloads using a prototype implementation of REWIND. Also, we demonstrate the temporal isolation of consecutive function requests handled within a shared function container.

The remainder of this paper is organized as follows. The following section explains the background and motivation of this paper. Then, we explain our approach in Section 3. Section 4 gives the performance and security evaluation results. We contrast our work with the related work in Section 5. We conclude this paper in Section 6.

2 Background and Motivation

2.1 Insecure Sandbox Reuse

Serverless computing has drawn strong attention in cloud computing since cloud customers (or application developers) can solely focus on their application development, and server-related burdens, such as server management, load-balancing, and auto-scaling can be offloaded to serverless computing frameworks (e.g., AWS Lambda) [26, 36]. Serverless computing also offers better pricing since customers are charged for the actual time spent by their invoked functions [36]. Due to these benefits, it is reported that many IT companies migrate their applications to serverless computing [31].

Serverless computing is known to provide strong security for two reasons. First, sandboxing is applied; a function is executed inside an isolated execution environment, such as containers or virtual machines (VMs) [1, 35]. As a result, the exploitation alone does not grant much security benefit to the attacker, as its privilege is strictly restricted within the sandbox. Second, the function execution is ephemeral. This reduces the attack scope in terms of time [56]. A function in serverless computing is stateless and can run on any server hosts in the cloud. An invocation of a function, or a function instance, is ephemeral; the function code and data are valid only while the function is being executed. After it completes, its state and data do not last but disappear. When applications need to persist their data, applications are supposed to store data in cloud storage services. This ephemeral characteristic allows to eliminate the *persistence* of function code and data, which otherwise can be a frequent source of security attacks. These two characteristics allow strong isolation of a function from other functions as well as inter-invocation of the same function. Industry practitioners believe the ephemeral characteristic can significantly reduce the possibility of security attacks because the life cycle of function instances is fairly short [13, 56].

However, the security of serverless computing in practice is sometimes compromised for better performance. So called warm containers or warm functions are the technique to improve the function invocation performance. Since the cost of initializing a container for function invocation is high, serverless platforms adopt to reuse function containers for the invocation of forth-coming function requests. Figure 1(a) shows the pseudo code of a function process that repeats handling function requests while the container is kept alive. This warm container technique can improve the performance of function invocation since the time to initialize the function instance is avoided. Unfortunately, the container reuse technique fails to provide strong isolation between different function requests and can open up attack opportunities [4, 13, 27].

Once a function container is reused, the data of the previous function invocation can persist in memory and/or in a temporary file system (e.g., ⟨/tmp⟩). This *quasi-persistence* of data can open up new security attack opportunities, turning an unexploitable vulnerability in a function into an exploitable one [4, 13, 27]. For example, attackers can exploit the vulnerability of function codes and leave rootkits in a function container. The successive function requests can be attacked by these rootkits. In addition, privacy-sensitive data left mistakenly can be exfiltrated by attacker's function request. These new attack opportunities are caused by sharing a function container for different function requests and failing to enforce isolation between them.

2.2 Threat Model

This work focuses on the security problem of serverless applications running on commercial public cloud platforms, such as AWS Lambda. We assume serverless functions handle the privacy-sensitive data of clients. A function container services a request of a client at a time, but the container hosting function execution can be reused to service function requests of other end users, hence a warm container. A container has room for persistence, such as the temporary file system, writable memory where function code is running, and functions can leave unencrypted privacy-sensitive data in such persistent storage. Our scheme does not assume each function code is malicious.

We further assume that an implementation of a function has a vulnerability. Exploiting this vulnerability, the attacker attempts to leak the privacy-sensitive data left in the function. Specifically, the attacker can exploit the vulnerability and leak leftover data inside the warm container's persistent points.

Our scheme does not assume any side-channel attacks to leak data. We assume the cloud provider, the serverless platform, and the computing infrastructure are trusted. Any privileged components, such as the operating system kernel and hypervisor, are also trusted and do not collude with attackers.

```
from code import func
do
  args = recv(proxy)
  result = func(args)
  send(proxy, result)
while keepalive == True:
```

(a) function process in warm container

```
from code import func
do
  args = recv(proxy)
  if (child = fork()) == 0:
    result = func(args)
    exit(result)
  else:
    wait(child, &result)
  send(proxy, result)
while keepalive == True:
```

(b) Request isolation using fork

```
from code import func
do
  /* checkpoint/restore point */
  args = recv(proxy)
  result = func(args)
  send(proxy, result)
while keepalive == True:
```

(c) Request isolation using checkpoint/restore

Figure 1: The pusedo code of a function handler: (a) warm container, using (b) fork and (c) checkpoint/restore.

2.3 Existing Solutions and Their Limitations

An intuitive but effective solution to providing strong security to serverless applications is eliminating the quasi-persistence of data during function invocation. At the same time, the solution needs to be performant, hence invoking a new container for every function request is not acceptable due to its high performance overhead. We can identify two possible solutions to this problem and illustrate the limitations of them.

Isolating virtual memory using fork. The simplest approach to eliminating the memory quasi-persistence is the use of the `fork()` system call. With the warm container technique, a function process in the function container runs a while loop and the loop body serves the function request as shown in Figure 1(a). In this approach, however, before invoking the function code, the function process forks and the child process actually serves the function request as shown in Figure 1(b). By doing so, the memory residue left by the execution of a function is removed after the child process exits. A consecutive function request is also served on a new child's virtual memory which contains no data left by the previous function invocation.

Limitations of fork. However, this simple fork-based approach has limitations. First, fork incurs copy-on-write handling overheads. When a child process modifies pages shared with the function process (the parent), the kernel page fault handler is invoked to perform copy-on-write. The page fault handling overhead can be exacerbated when a forked process contains file pages or zero pages as the current fork implementation does not copy page tables for such memory pages [10], which accompanies additional page faults. In addition, frequent handling of page faults can incur hidden performance overheads caused by architectural resource pollution [23, 54]. Even worse, this CoW overhead repeats on every function invocation. Second, the current implementation of fork (e.g., in Linux) does not support the fork of a multi-thread process. Function codes are implemented in multi-threads for high performance [4, 61]. Lastly, fork cannot eliminate quasi-persistence of files [13]. Function containers provide a temporary directory to store temporary files of function codes. Hence, the quasi-persistence of files needs to be supported but fork provides only the memory isolation. Consequently, the fork-based approach is incomplete in providing isolation between function requests.

Checkpoint and Restore (C/R). C/R is a viable approach to providing isolation between function requests. A snapshot of a function process is taken and is used every time to serve a function request. Accordingly, C/R can enforce isolation between function requests. CRIU [11] is a representative implementation of C/R in Linux systems. By using CRIU, memory and file quasi-persistence can be removed. However, CRIU is a general implementation and hence incurs high overhead of (de)serializing snapshot of a process [4, 60]. Groundhog [4] is a light-weight implementation of C/R specialized to serverless computing framework. Specifically, Groundhog takes memory snapshot of a function process. Then, it uses the Linux's soft-dirty feature [55] to track dirty pages during function request serving [4, 11]. After a function process handles a function request, its memory state is restored from the snapshot; the original values in the snapshot are dumped to dirty pages. Therefore, any privacy-sensitive data in memory can be removed.

Limitations of C/R. However, the C/R-based approaches has the following four limitations. First, C/R doubles memory consumption for managing the snapshot of a function process. Since the C/R-based approaches are performed at user-level, the memory snapshot captures the entire virtual memory. When restoring memory from the snapshot, all the original data of the entire virtual memory should be necessary. The read-only mappings cannot be trusted at user level since attackers can change the mapping permission (e.g., `mprotect()`) to writable, plant any attackable data, and restore the original read-only permission. Second, the C/R-based approaches incur high overheads during checkpointing and restoring. Since the snapshot is managed at user-level, it incurs extra overheads of syncing the kernel page table with another one that is managed at user-level for a snapshot. During checkpointing, all the page table entries (PTEs) are copied to the user-level to take the snapshot of the virtual memory state. During restoring, all the PTEs are copied again to inspect dirtied or modified PTEs, which will follow the restoration of the original data by comparing them to those taken during checkpointing. Third, tracking dirty pages incurs runtime overheads of exception handling. The soft-dirty feature makes use of the page fault handler of the kernel [55], which frequently interrupts user code execution with kernel code interventions [23, 54]. Last, taking a memory snapshot in user space requires administrator capability (e.g., `CAP_SYS_ADMIN` in docker) since it needs to access special files (e.g., `/proc/pid/mem`, `/proc/pid/pagemap`, etc.) [4, 21]. Assigning the administrator capability to a container can be dangerous since once the container is compromised, the entire system can be exposed to attackers.

2.4 Challenges

Performance overheads. Providing isolation between function requests, thereby eliminating the quasi-persistence is crucial for building secure serverless platforms. The methodology of providing isolation also needs to be performant considering short function execution times. For example, typical serverless functions take 50 milliseconds of execution time and if the C/R-based approach adds up 10 milliseconds of extra latency, the function throughput can be degraded by 20%, which is not negligible. Additionally, the runtime overheads, such as frequent page faults caused by copy-on-write protection, may exacerbate the function execution performance. Hence, it is necessary to minimize such performance inhibitors.

Memory consumption. The resource consumption also needs to be minimized. The C/R-based approaches need to capture the entire virtual memory of a function process. This amplifies the memory cost of providing the protection. The reason to capture the entire virtual memory is that the system does not know which pages will be modified. Since the memory snapshot is managed at user-level, it is safe to keep the original values of the entire virtual memory. Even read-only pages can be modified by altering the protection of page mappings to writable. The fork-based approaches can minimize the memory consumption since they can precisely capture page modification (i.e., page faults). However, the page fault overhead is high. The exception handling overhead is not negligible due to architectural resource pollution by kernel code execution [23, 54] as well as the cost of architectural vulnerability mitigation features [24, 50] (e.g., kernel page table isolation [30]). High memory consumption can also affect the overall function serving performance of serverless clusters. Warm containers can be replaced due to memory shortage [19, 53]. Then, the high-overhead cold start of function containers may affect the function execution performance. Since the number of warm containers is largely dependent on the memory consumption, it is necessary to keep memory consumption low.

Missing Persistence Points. Memory is not the only persistence point in serverless function containers. A temporary file system (`/tmp`) is provided in a function container and any privacy-sensitive data can remain in that storage. The temporary file system is generally a RAM file system hence volatile but is persistent while the function container is alive. Any data left in the file system can be attack targets. Similarly, processes or threads can also be a persistence point and can be exploited. Attackers can plant a rootkit or malware process in a function container to attack other function requests. Since the aforementioned approaches only sanitize memory, this type of attack cannot be prevented properly. Therefore, files and processes/threads also need to be carefully addressed to eliminate quasi-persistence in warm containers.

3 REWIND

3.1 Overview

This paper proposes REWIND, a secure and fast function execution platform. REWIND provides temporal isolation between function requests served on the same warm function container. This is achieved by (1) taking a snapshot of the container without any private data and then (2) rolling back the container state to the snapshot state at the end of each function serving.

To achieve high-performance function execution, our scheme exploits the temporal locality of function code execution in a function container. A warm container serves consecutive function requests of the same function. In other words, the runtime behavior of the function process is (almost) identical across different runs. For example, memory buffers allocated in the current run can be allocated again in the next run. Pages made dirty in the current run may be dirty in the next run.

To achieve low memory consumption, REWIND has kernel-level support to manage a memory snapshot. REWIND exploits CoW-based page sharing to save memory and keep a snapshot isolated from the current memory of a function process. Meanwhile, REWIND leverages the repetitive execution of functions for performance. When frequent CoW-breaking is detected, such pages are duplicated. If new memory mappings are made repetitively in the middle of function execution, such memory areas are buffered and used in the next run to avoid costly page faults and page allocations.

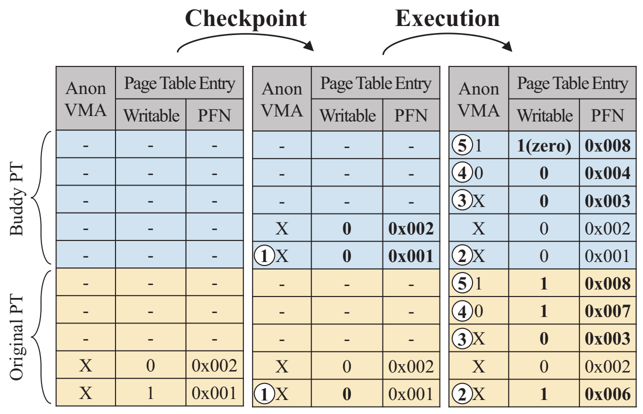
To this end, REWIND proposes two privileged operations `snapshot()` and `rewind()`. A serverless function container has two processes: a *proxy process* and a *function process*. The proxy process acts as a proxy for the function container, relaying requests/responses between the serverless framework and the function container. The two privileged operations are invoked by the proxy process to handle the function process. When a function container is initialized and a function process is created, the function is invoked using a *dummy* argument to initialize its relevant codes, data, and potentially soft states, such as cached data and memorization [4]. Then, the proxy process takes a snapshot of the function process. After that, the function process handles regular function requests. Whenever the function process finishes serving a function request, its result is forwarded to the proxy process, which then invokes the rewind operation to roll-back the state of the function process to its snapshot state.

A function container can have three quasi-persistence points: memory, file system, and processes. The following subsections describe how REWIND addresses and eliminates quasi-persistence in these points.

3.2 Memory

Memory is the most important resource to be carefully handled in REWIND. This is because most user-private data resides in memory, and its amount is the largest among the three persistent points, which mostly affect the performance of private data cleaning. Our approaches are (1) taking a snapshot of memory, (2) tracking modification of memory, and (3) leaving necessary information in the snapshot for fast rewind operation. To facilitate these approaches, we introduce *buddy page table*, an extended version of conventional page table that manages two virtual memories, the original one and the snapshot. The buddy page table is designed to contain necessary information for taking a snapshot, rewinding the virtual memory using the snapshot, tracking written pages, and accelerating the reuse of dirty pages.

With the buddy page table, the leaf page table size is increased from 4 KB to 8 KB. The lower 4 KB is the original page table that stores PTEs for the original virtual memory. The upper 4 KB serves as the buddy page table for the lower 4 KB original page table, storing buddy PTEs that manage the snapshot of the original virtual memory and related information. The memory management unit of the CPU accesses only the lower 4 KB page tables, while the upper page tables are managed only by software. The snapshot is carefully manipulated by the snapshot operation and page population operations, such as memory mapping (`mmap()`) and page fault handling. The simple 8 KB page table structure facilitates accesses to the regular PTEs and their associated buddy PTEs during such memory manipulation operations. In REWIND, only the function process is allowed to have a buddy page table, and other processes are not.



	Anon VMA	Page Table Entry	
		Writable	PFN
Buddy PT	-	-	-
	-	-	-
	-	-	-
	-	-	-
	-	-	-
Original PT	-	-	-
	-	-	-
	-	-	-
	X	0	0x002
	X	1	0x001

	Anon VMA	Page Table Entry	
		Writable	PFN
Buddy PT	-	-	-
	-	-	-
	-	-	-
	X	**0**	**0x002**
	①X	**0**	**0x001**
Original PT	-	-	-
	-	-	-
	-	-	-
	X	0	0x002
	①X	**0**	0x001

	Anon VMA	Page Table Entry	
		Writable	PFN
Buddy PT	⑤1	**1(zero)**	**0x008**
	④0	**0**	**0x004**
	③X	**0**	**0x003**
	X	0	0x002
	②X	0	0x001
Original PT	⑤1	**1**	**0x008**
	④0	**1**	**0x007**
	③X	**0**	**0x003**
	X	0	0x002
	②X	**1**	**0x006**

Figure 2: Memory snapshot example. (X indicates don't care)

3.2.1 Snapshot

The memory snapshot operation takes a snapshot of the current process and prepares memory write tracking. Our scheme makes use of the well-known copy-on-write (CoW) technique to track memory writes and to preserve the original snapshot while minimizing memory space overhead.

Basically, the snapshot operation applies to memory regions (or virtual memory areas (VMAs)) mapped already, say *committed* VMAs. Hence, any memory regions created after snapshotting, say *uncommitted* VMAs, are not considered to be in the snapshot since they can contain privacy-sensitive data. For performance optimization, we handle such new VMAs in a different way; see Section 3.2.3 for details.

Figure 2 shows an example of the memory checkpoint operation and the state of the page tables. Committed VMAs have two types of pages, populated (or present) pages and non-populated (or non-present) pages. For populated pages, we clone the PTEs of the original page table to the buddy page table and apply the copy-on-write protection to both PTEs. If a page is read-only, this approach saves memory space. If a page is writable, the CoW protection may save memory space or preserve the original copy of the page even if the page is written thereafter. In the figure, the two PTEs are present in memory and are cloned after snapshotting. The cloned PTEs (or buddy PTEs) have the same page frame number (PFN) of the original PTEs. The PTE at the bottom was writable before snapshotting but is now write-disabled (CoW-protected) ①. Note that these operations are applied to anonymous VMAs as well as private file VMAs (e.g., data sections of executable or shared libraries).

When writes occur on such CoW-protected pages, we apply the conventional CoW handling to them ②. Hence, the original PTE now references a new page and accommodates memory writes; this may contain privacy-sensitive data. At the same time, the buddy PTE points to the original copy of the page to keep the snapshot correct.

For present pages in a shared file mapping, we do not protect writes on them. Writes onto shared file pages may contain privacy-sensitive data. Nevertheless, our memory rewind operation reverts file data back to its pristine state by our file system snapshot-rewind operation, which is explained in Section 3.3. The PTEs in a shared file mapping reference page cache pages. Hence, if the file system rewind completes, the reverted file data is immediately visible to a process under the Unix semantics [32]. Therefore, the memory snapshot-rewind is not concerned with shared file pages.

For non-present pages, their snapshot operation occurs when those pages are populated (i.e., page faults). For read faults, the identical CoW protection is applied to pages after read faults are handled ③. However, for write faults, we conduct two different operations to file-backed private pages [1] and anonymous pages. For file-backed pages, the fault handler performs the conventional filemap fault and then, our scheme applies the CoW breakage to the page. Hence, the buddy PTE points to the original copy of the file page (i.e., page cache page), and the original PTE points to the private copy of the file page ④.

However, for anonymous pages, when their write faults occur, the pages are not handled under the CoW semantic. Since we know the original content of each page, which is zero-filled, we make both PTEs share the same physical page. Then, we set a special *zeroing* flag in the buddy PTE to indicate that its original value is zero ⑤. This flag is used later during the rewind operation.

3.2.2 Rewind

Memory rewind restores the memory of the function process to a pristine state using the snapshot. This operation refers to the contents of the buddy page table and restores the process's original virtual memory as follows.

For committed VMAs, the buddy PTEs reference one of the following cases: (1) a page shared with the original PTE with CoW protection (write-disabled), (2) an individual page not shared with the original PTE (i.e., CoW occurred), (3) a page shared with the original PTE but containing the special zeroing flag, (4) a file page in a shared file mapping. Note that the memory rewind operation covers the former three cases, and the last one is covered by the file system rewind (Section 3.3). Figure 3 shows an example of changing the page table state during memory rewind.

In the first case, the memory rewind operation does nothing because the original PTE points to an unmodified page whose contents are identical to the contents of the snapshot ⑥. This is because the virtual page has not been modified since the snapshot was taken. Both PTEs are still protected by CoW (i.e., read-only), which means that any writes to the virtual page may preserve the original data of the snapshot. Therefore, the memory rewind operation cares nothing in this case.

[1]The file-backed private pages are created when pages in private file mapping are modified. Examples are writes onto data sections of executable or shared libraries.

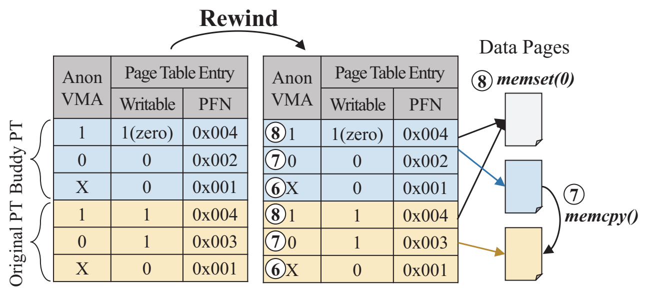


Figure 3: Memory rewind example.

In the second case, we copy data from the buddy PTE's page to the original PTE's page. This happens because the CoW protection is broken by writes to this virtual page. Therefore, we need to clean up the writes since the snapshot was taken. An intuitive approach is to free the original PTE's page and make the original PTE reference the buddy PTE's page with CoW protection. However, this intuitive approach comes at the cost of repetitive page faults when the function process runs again to handle another function request. Our approach is to preserve both pages of both PTEs and copy the original content of the page back to the original page ⑦. In this way, the rewind operation correctly restores the original content of the virtual page. In addition, no further page faults occur on this type of page. Given the temporal locality of function execution, this type of page is written repeatedly during function execution. For example, once a global variable is modified in a previous function execution, the variable is likely to be modified again in the next function execution. Consequently, we reduce page faults by preserving both pages and allowing write access on the rewind operation.

In the third case, the original PTE is preserved and the page is cleared to zero. This case occurs when anonymous memory, such as stack and heap, has grown and pages in it are modified. Accordingly, we reset the contents of such pages to the original, which is zero. This case also exploits the temporal locality of function execution. For example, if a stack has grown to a certain point in the previous function execution, the stack is likely to grow to the same point again. Therefore, we preserve these pages to save time that would be spent on page allocation and page fault handling.

3.2.3 Handling Uncommitted VMAs

During function execution, any memory areas (or VMAs) can be mapped after the snapshot is taken. We call such memory regions uncommitted VMAs. Since uncommitted VMAs do not belong to the snapshot, which means that such VMAs may contain privacy-sensitive data, it is intuitive to discard them all if they exist on memory rewind. Unfortunately, this simple approach may give up some performance optimization opportunities that result from repeatedly running the same function. For example, during function execution, the function code requests to open a new memory mapping for its memory allocation. It is then likely that an identical memory mapping will be required when the function process rewinds and executes the same function code again.

Our optimization approach is to reuse such uncommitted VMAs. During the memory rewind operation, such VMAs are unmapped but their resources (i.e., pages and page tables) are not freed, and are instead preserved for reuse. To prevent private data leakage, pages are cleared to zero. When the function process rewinds and handles another function request, if certain conditions are met, the preserved VMAs are reused to reduce the time spent establishing new memory mappings. To this end, during the memory rewind operation, anonymous uncommitted VMAs are detached from the virtual memory. In other words, the VMA structure is detached but preserved, along with PTEs and pages. After rewind, when a similar anonymous memory mapping occurs, the arguments of the new memory mapping are compared with the preserved ones. Specifically, a new mmap system call without a designated base address and with identical mapping flags is considered for VMA reuse. We use a best-fit algorithm to match and reuse the preserved VMAs. If the reused VMA is larger than a requested mmap, the surplus area is preserved for further reuse. If the reused VMA is smaller than a requested mmap, remapping is performed to increase the size of the reused VMA.

Note that we limit the target of this VMA reuse to anonymous memory. If the target is expanded to include file-backed memory, it can be the source of privacy leakage. Or, even if a file does not contain any privacy-sensitive data, the file information, such as the file name, can also be leaked as a method of side-channel attacks. Therefore, we restrict the VMA reuse to anonymous memory.

3.3 File System Snapshot-Rewind

File system snapshot is also necessary since serverless platforms allow file writes in an ephemeral sandbox. Hence, privacy-sensitive data can remain in ephemeral storage and can be leaked by attackers. Applying the snapshot-rewind operation to the sandbox file system, however, is not a big issue since taking a file system snapshot is a well-supported feature of modern file systems [40, 51] or databases [28]. It is straightforward to use such file systems for our scheme, taking a snapshot of the file system during snapshotting and restoring the snapshot during rewinding.

Nevertheless, the file system used in the serverless platform we are based on does not support the snapshot feature during runtime. The file system of the container-based sandbox (e.g., docker [16]) is OverlayFS [43], which is a union file system of multiple file systems. This file system is effective for containers (e.g., docker) with baseline image [65]. The baseline image is represented as a *lower file system* in OverlayFS. Any modification made during runtime is recorded in a file system called *upper file system*. The processes inside a container see

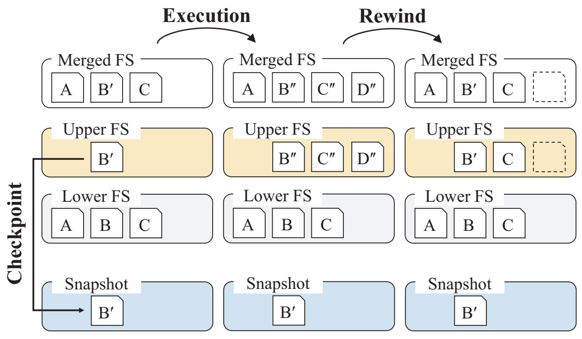


Figure 4: Example of file system snapshot-rewind.

only the merged view of the two file systems (lower + upper). The lower layer is immutable while the upper layer is volatile. The union of the two file systems can provide an ephemeral file system of a baseline container image; hence facilitating the deployment of a function image to many hosts [52].

We implement the snapshot-rewind operation on OverlayFS as follows. During the snapshot operation, we take a snapshot of the upper file system and store them separately, which is denoted as a snapshot file system. Then, when the rewind operation is invoked, we restore the content of the snapshot onto the merged file system. We take this approach because OverlayFS does not allow direct modification or remount of the upper file system [44]. Figure 4 shows an example of our file system checkpoint-rewind operation.

In more detail, during the rewind operation, we perform the following procedure on all the files in the current upper file system. For each target file in the upper file system, we first find the original file of the target file. If the original one is found either from the snapshot or from the lower file system, we compare the `mtime` of both files. If the target file (B'' or C'') is younger than the original one (B' or C), we replace the target file with the original one. If the original one is found from both file systems (target B''), we select the original one from the snapshot (B'). If the original one is not found, we delete the target file (D'') from the merged view. Considering that OverlayFS does not allow direct change of the upper file system, we believe this approach is a reasonable practice to implement the checkpoint-restore operation on OverlayFS. Note that our procedure can be inefficient since all the operations are not done at file system-level but at the user-level. However, our evaluation results show that the overhead of file snapshot-rewind is negligible since the data handled during file system snapshot-rewind is only a few tens of KB. We left the file system-level implementation of the snapshot-restore, which can be more efficient than our practice, as the future work.

3.4 Task Snapshot-Rewind

REWIND rewinds tasks (processes and threads) at every completion of a function execution. Otherwise, the attacker's threads/tasks can remain in a container for malicious behaviors, such as rootkits [13]. Hence, we record processes and threads in a function container when the snapshot operation is invoked. Recall that the serverless platform and privileged components (container and the OS kernel) are trusted. Hence we assume no malicious ones exist while doing the first snapshot operation, allowing to take the snapshot of tasks. During the rewind operation, we kill any tasks that are not in the snapshot. Any memory spaces, such as thread stacks and process virtual address spaces, are also reclaimed during this operation. We do not explicitly release resources, such as futex, semaphore, or file lock, that are held by killed tasks. If the killed components hold such resources, they would have not worked properly with the original warm sandbox technique.

REWIND protects the function process against malicious use of the snapshot operation. If the snapshot operation is exploited by attackers, the snapshot operations would be nested. However, we assume the first call of the snapshot operation is trustworthy since the serverless platform and privileged components are trusted. Hence, the first snapshot operation is a proper one. During rewind, our scheme trusts only the snapshot made by the first snapshot operation. Hence, even if the snapshot is nested the function process can return to its proper pristine state.

Whenever the rewind system call is invoked, any open files and open network connections also need to be rewound. Otherwise, they would incur incorrect execution, e.g., reading a file from an incorrect position, or using unauthorized communication channels. Such file descriptors and network connections are task-private. Hence, we record the open file descriptors and their current positions during the task snapshot operation. During the rewind operation, we close any file descriptors that were opened after the snapshot operation. We also reset the seek offset of the file descriptors if their back-end is a file.

4 Evaluation

4.1 Evaluation Environment

The memory/task snapshot-rewind operations are implemented as new system calls in Linux kernel version 5.4.0. The file system snapshot-rewind operations are implemented as user-level commands. All the snapshot-rewind operations are invoked by the proxy process of the OpenWhisk serverless framework [42,45,46]. The snapshot/rewind points of the function process are identical to those of the C/R approach in Figure 1(c). We modified 977 lines of Linux kernel code and 363 lines of OpenWhisk code. Experiments were conducted on a server with an Intel Xeon Gold 5118 processor (2 sockets, 12 cores per socket) and 192 GB of RAM.

We built and used three microbenchmark programs. *Memory* examines the performance characteristics of REWIND and other schemes using simple memory access operations on memory mapped using `mmap()`, varying in memory size, type, and read/write ratio. *Hello* and *PKG* only import libraries and

Microbenchmark	**Memory**: read/write mmap'ed memory **Hello**: Hello world printer **PKG**: Package importer (mypy, numpy, django)
Float	Floating point operation
MatMul	Two N-dimensional square matrix multiplication
Linpack	Solving linear equations
PyAES	AES encoding
ImageProcessing	Image transformation using Python Pillow library
VideoProcessing	Video transformation using Python OpenCV library
Chameleon	HTML generation
Machine Learning	**LR-Training**: Review analysis **LR-Serving**: Review analysis

Table 1: Workload specification (workload names bold-faced).

do not have a function body. Also, we chose workloads from the FunctionBench benchmark suite [29]. The workloads used in our experiments are summarized in Table 1. Although the tested function workloads are written in Python, REWIND is not limited to a specific runtime and can be applied to any runtime or native code. REWIND is compared with the following alternative schemes:

- *Base*: This scheme represents the original OpenWhisk utilizing the warm container technique. The function handler operates as depicted in Figure 1(a).
- *Fork*: This approach employs the `fork()` system call for memory and task isolation, as shown in Figure 1(b).
- *Groundhog (GH)* [4,21]: This is the state-of-the-art method for ensuring isolation between function requests. It creates a snapshot of a function process after handling a dummy request. Then, before serving a new function request, the memory state is restored from this snapshot. The function handler code is as shown in Figure 1(c).

4.2 Memory Microbenchmark

We first demonstrate how REWIND and the compared schemes perform in various memory access scenarios using the memory microbenchmark. The microbenchmark program performs the following operations: (1) allocating a memory using `mmap()`, (2) populating all the pages in the memory by writing one byte on each page, (3) conducting the benchmark operation by reading or writing the first byte of the pages sequentially. For REWIND and GH, the snapshot is taken after operation (2). For Fork, `fork()` is invoked after operation (2). After operation (3), REWIND and GH perform their restore operation. Fork simply terminates a child process. The performance metrics are *function* time, which is the execution time of operation (3) and a *restore* time which is the time taken to restore the memory of the benchmark process from its snapshot.

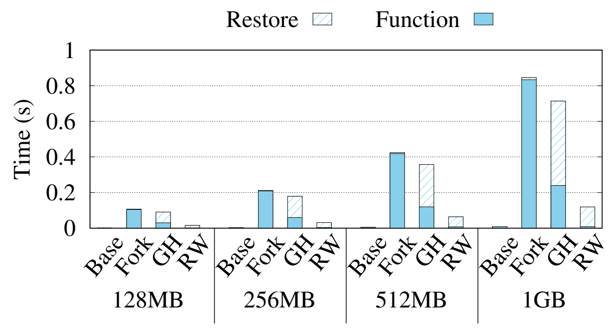


Figure 5: Microbenchmark performance with varying memory size.

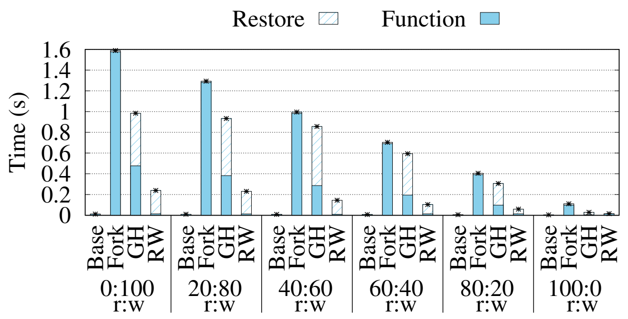
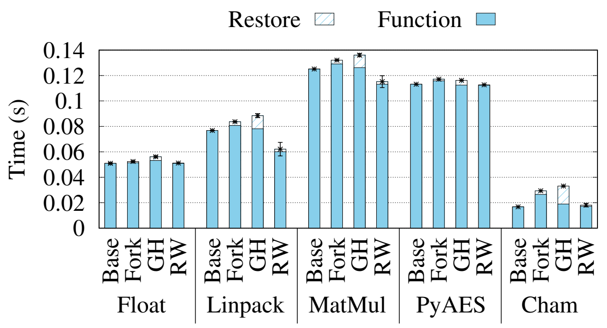


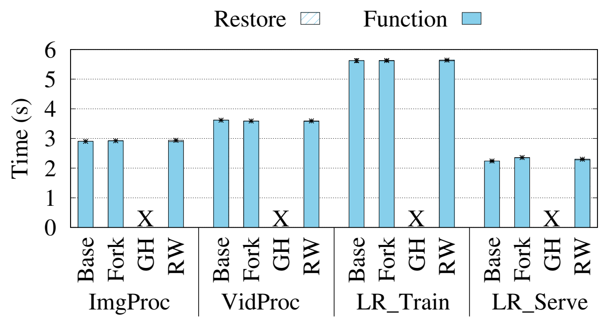
Figure 6: Microbenchmark performance with varying ratios of read/write operations.

Figure 5 shows the performance of the workload with varying the size of memory. In this workload, the memory is a private file mapping, with sizes varying from 128 MB to 1 GB. The memory operations are reads and writes in a 1:1 ratio. As shown in the figure, function time increases as memory size increases. More importantly, GH and Fork show high function time overheads as they require page fault handling on every first page modification. Fork shows the highest overhead as its fault handling requires a page allocation followed by a page copy. GH shows a moderate overhead as its page fault handling requires only setting a soft dirty bit. REWIND, however, shows the lowest overhead in function time because it does not cause page faults as the write permission of the memory is enabled. In terms of the restore time, GH shows higher overhead than REWIND as its restore operation occurs in user space. However, REWIND performs the restore operation in the kernel, simply performing memory copies from buddy pages to original pages.

Figure 6 illustrates the performance of the microbenchmark with varying ratios of read/write operations on a 1 GB memory. The overheads associated with the function and restore times can be largely affected by the portion of dirty pages. This performance characteristic is demonstrated in this experiment as shown in the figure. When the portion of memory writes decreases, the function time and the restore time decrease. Among the three schemes (Fork, GH and REWIND), REWIND shows the lowest overhead as it shows the shortest



(a) Memory Workloads



(b) File Workloads

Figure 7: Function latency with restore time. X indicates no result because GH failed to execute the workloads.

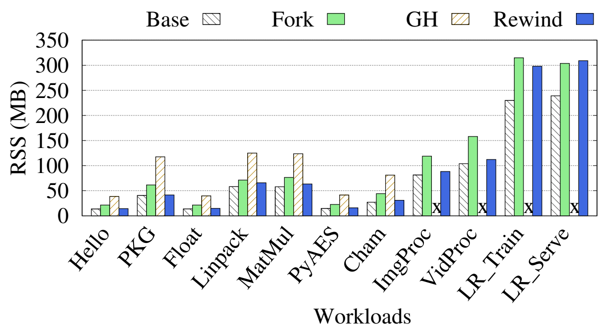


Figure 8: Peak RSS of function containers.

function and restore times.

4.3 Function Latency

To understand the performance impact of REWIND on real serverless functions, we measure the function latency of the workloads in FunctionBench. The function latency is the latency from when the proxy process receives a function request to when the proxy process obtains the result from the function handler process. Figure 7 shows the function latency results of the tested workloads.

First, Figure 7(a) shows the latency of workloads with fast function execution time around 10s of milliseconds. As shown in the figure, REWIND shows similar performance to the baseline despite that REWIND performs the rewind operations. Moreover, REWIND demonstrates shorter function execution times than the baseline in the *Linpack* and *MatMul* workloads through the benefit of the VMA reuse technique, ultimately showing 19% and 11% shorter latency than the baseline, respectively. Meanwhile, GH incurs a larger restore overhead (11% on average) compared to REWIND, particularly with the workloads (Linpack, MatMul, Cham) using a large amount of dirty anonymous pages. GH causes page faults during its function execution by its soft-dirty feature, resulting in a 3% increase in function time compared to the baseline. Fork experiences a 12% increase in function time compared to the baseline due to its copy-on-write handling overhead during execution.

Figure 7(b) indicates that function times of file workloads are longer than one second. Interestingly, these workloads manipulate files, and GH has malfunctioned with these workloads. Therefore, their results are omitted and marked as 'X' in the figure. As shown in the figure, the page fault overhead of fork is hardly noticeable due to long execution times. Only *LR_Serve* shows a slightly longer execution time than the baseline. REWIND also shows marginal performance overhead compared to the baseline. Note that among the four schemes, only REWIND enforces the isolation to the three persistence points (memory, file, and tasks).

4.4 Memory Consumption

Figure 8 illustrates the memory consumption for each of the four schemes, measuring the peak resident set size (RSS). The figure shows that the schemes providing isolation, generally, consume more memory than the baseline due to the management of duplicated copies of dirty pages. Notably, the GH scheme exhibits the highest memory usage as it requires snapshotting all the pages of the function process. The Fork scheme, which duplicates dirty pages during function execution, shows the second-largest memory consumption. In contrast, REWIND demonstrates the most efficient memory usage among the schemes addressing security issues in warm containers. Its memory overhead is, on average, only 11% higher than the baseline. This efficiency is attributed to REWIND's design, where anonymous pages are not duplicated between the snapshot and the function process, as detailed in Section 3. It's important to note that this approach does not compromise privacy data security; anonymous pages are reset to zero during each rewind, ensuring no leakage of sensitive data.

4.5 Function Throughput

The performance of function handling is influenced not only by the execution time of the function itself but also by the memory consumption of the function containers. To evaluate

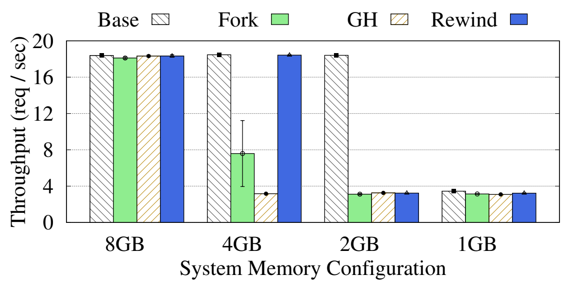


Figure 9: Function throughput

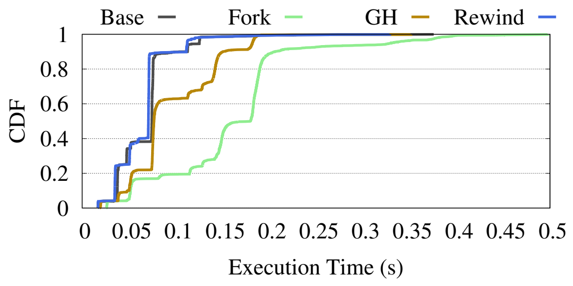
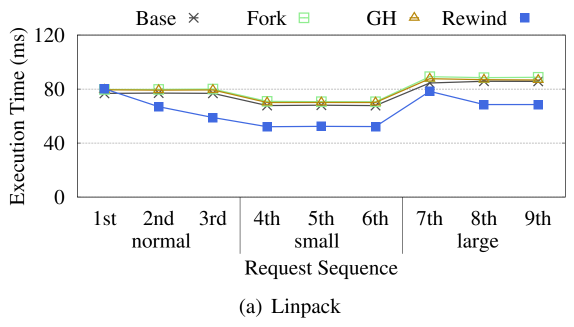


Figure 10: CDF of function execution time. System memory configuration is 4 GB.



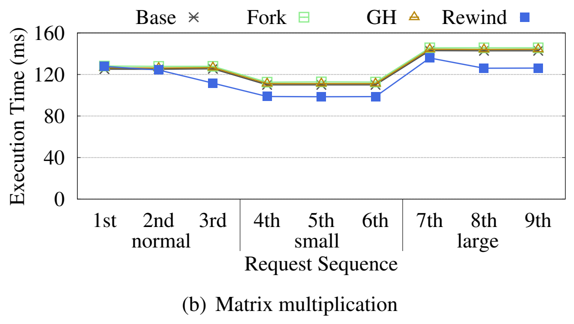


Figure 11: Function run-to-run execution time with varying function arguments.

function execution performance thoroughly, we set up a function execution platform and ran a trace of function requests. This trace was collected using the Microsoft Azure serverless function trace [63]. From this trace, we extracted a sequence of function requests over a 20-minute period, mapping each function ID to our tested workloads. We identified 14 unique function IDs and mapped two to three functions to each of our tested workloads. We excluded workloads that use files, as Fork and GH do not support the elimination of file system persistence.

Each function container's memory allocation was configured to accommodate the peak RSS depicted in Figure 8. The available memory sizes for containers were 32 MB, 64 MB, and 128 MB. For instance, if the peak RSS of MatMul in REWIND is 60 MB, a container with 64 MB of memory would be assigned to that workload. A node was set up with the OpenWhisk platform to run function containers to handle the function traces. The OpenWhisk platform was configured to utilize the warm container technique as much as possible, except in cases of memory shortages. In situations where memory was insufficient, the least recently used container would be terminated to free up memory and accommodate the current function request.

Figure 9 shows the throughput of the function trace with varying the available memory of the node. As shown in the figure, when the memory is sufficient (8 GB), all the schemes show high function execution throughput (around 17 function requests per second). When the available memory is reduced to 4 GB, Fork and GH show poor performance due to memory shortage followed by killing and cold-starting function containers. However, REWIND shows good performance which is comparable to the baseline, because of REWIND's low memory overhead. When the available memory is reduced to 2 GB or 1 GB, the throughput drops as the memory shortage incurs frequent killing and cold-starting of function containers.

Figure 10 presents the cumulative distribution function (CDF) of the function execution times for the workload. Each function execution time represents the end-to-end latency from requesting a function execution to receiving the function response. This includes the time spent to rewind/restore the function handler process. As depicted in the figure, with 4 GB of memory, REWIND exhibits a function execution time similar to that of the baseline. However, Fork and GH show an increase in function execution time due to the more frequent occurrence of cold container invocations.

4.6 Benefits of VMA Reuse

REWIND optimizes function execution by reusing anonymous VMAs. To test whether this feature adapts to changes in function arguments, we measured the performance of Linpack and MatMul; these two workloads benefit from the VMA reuse feature. Figure 11 shows the execution time for these two workloads with three different sets of arguments (base, small, large), each tested in three consecutive runs. Therefore, each workload undergoes a total of nine consecutive function invocations. As illustrated in the figure, the baseline, Fork, and GH show no significant differences across varying func-

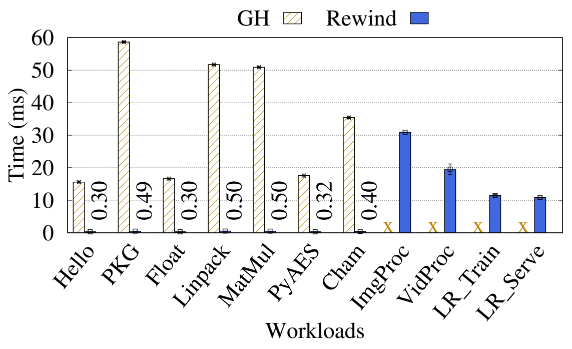


Figure 12: Snapshot creation time. X indicates no result because GH failed to execute the workloads.

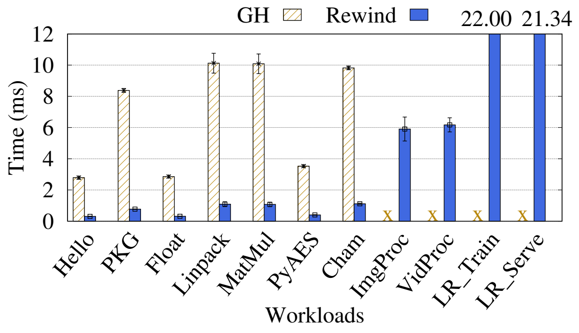


Figure 13: Restore time. X indicates no result because GH failed to execute the workloads.

tion arguments. However, REWIND demonstrates reduced function execution times when VMAs can be reused from previous executions. Notably, when the same-size arguments are used consecutively, REWIND further reduces the execution time compared to when handling different-size arguments. For instance, when the argument size changes from small to large, the performance gain of REWIND diminishes at the 7th run. However, REWIND regains its performance advantage when the same-size argument is used consecutively, as observed in the 8th run.

4.7 Time Overhead of Snapshot and Rewind

Figure 12 illustrates the time required to take a snapshot in REWIND and GH. This overhead is a one-time cost and, as such, has minimal impact on overall function execution performance. However, it can influence the cold start time of a function container. Since REWIND involves minimal operations during its snapshot operation, it outperforms GH in terms of snapshot creation time. For the memory-only workloads (Hello through Cham), REWIND records a maximum snapshot creation time of only 0.3 milliseconds, whereas GH ranges from 16 to 59 milliseconds. In the case of file workloads (ImgProc through LR_Serve), REWIND's snapshot creation time goes up to 30 milliseconds. However, this overhead is still negligible, especially when compared to the cold container launch time, which is on the order of hundreds of milliseconds [53]. The snapshot time of REWIND for the file workloads is mostly affected by the workload's file size. ImgProc generates 11 files with a total of 9 MB. VidProc generates 3 files with a total of 2.5 MB. LR_Train and LR_Serve generate one 51 KB file each. Note that GH does not support the elimination of file system persistence points. Hence, its evaluation with the file workloads is omitted.

```
import json, os, sys
def lambda_handler(event, context):
  name = event['name']
  os.popen("echo "+name+" >> /tmp/name.txt")
  res = os.popen("cat /tmp/name.txt").read()
  return {
    'statusCode': 200,
    'body': json.dumps(res)
  }
```

Figure 14: The example code for task and file persistence

```
name = abc » /tmp/name.txt; echo \"while :; do
echo 1 » /tmp/hello.txt; done\" > /tmp/t.sh;
chmod +x /tmp/t.sh; /tmp/t.sh &
```

Figure 15: Example input string exploiting the vulnerability of the function code in Figure 14.

Figure 13 compares the time taken to restore the snapshot of the function handler process in REWIND and GH. GH takes a long time to restore the memory of the function handler for two main reasons. First, the user-level implementation incurs high overheads. Second, and notably, it necessitates restoring all the pages in the virtual memory of the snapshot. However, REWIND benefits from lower overheads due to its kernel-level implementation and simplified restore operations. REWIND primarily copies back the original contents of the virtual memory pages from the buddy page table. The restoration overhead for REWIND is only 10.7% of that of GH. For the file workloads, the file rewind operation hardly affects the total rewind operations as the RSS of the workloads is much larger than their file sizes to rewind.

4.8 File Security Evaluation

The final part of our evaluation focuses on whether REWIND successfully eliminates file persistence, an issue that other schemes like Fork and GH do not address. To assess this, we ran the code depicted in Figure 14 and monitored for any residual data at the persistent points.

The function shown in Figure 14 includes a vulnerability that could exploit two persistent elements: tasks and files. For example, using the argument name from the example string in Figure 15, the code might unintentionally create and execute a shell file (`/tmp/t.sh`), resulting in continuous writing to `/tmp/hello.txt`. However, with REWIND, after executing the function code using the argument, no traces are left in the sandbox's file system or in the background tasks. This demonstrates REWIND 's capability to effectively remove both persistence points—the file system and tasks.

5 Related Work

Snapshot Function Sandbox Booting. Many studies have proposed to reduce the startup latency of a function container by exploiting the snapshot approach [5,59]. The use of a snapshot can avoid container initialization time. REAP [59] and Faasnap [5] have improved the cold startup time of a container by characterizing and prefetching pages actually used during snapshot booting. Although these approaches make use of a snapshot, their goal is not to provide isolation, but to accelerate the booting time of a function container. Such snapshot boot methods are usually integrated with microVMs [5,59], which are lightweight VM-based sandboxes [1]. REWIND is orthogonal to these approaches. A snapshot can contain REWIND's feature, which always rewinds the function's state to the pristine state before serving a function request.

Serverless Security. A few studies tried to address the security vulnerabilities of serverless computing [12, 13]. Valve [12] proposed to prevent privacy-sensitive data leakage by tracking taints and allowing them to pass through only authorized communication paths. ALASTOR [13] extends serverless security by auditing behaviors of the entire functions/sandboxes distributed servers. A few studies have proposed to build secure containers from privileged components using Intel SGX [3,8,49]. REWIND's purpose is orthogonal to confidential computing (e.g., Intel SGX), which would further strengthen the security of REWIND if combined.

Secure Sandbox. Enforcing isolation in a shared sandbox is important and many studies proposed to provide isolation across various resources [2,25,41,64]. Chancel [2] proposed multi-client isolation in a single enclave using per-thread memory region and software fault isolation. This work is similar to our work in terms of enforcing isolation between multiple end users. However, this work assumes the concurrent handling of multi-end users in a single container, which is not allowed in serverless computing. TxBox [25] exploited system transactions for secure execution in a sandbox. It detects and aborts any system transactions by malicious users using the TxOS's system transaction feature [48]. This work preserves privacy-sensitive data created or modified by allowed system transactions, which do not prevent persistence properly, but possible in REWIND. PRIVEXEC [41] allows private execution in a shared sandbox by providing secure and isolated storage access. This work is limited to preventing the persistence of a file.

Checkpoint and Restore. Taking a snapshot of a file system is supported in various file systems, including btrfs [51] and ocfs2 [40], as well as user-level tools, such as git [20]. Checkpoint and restore (C/R) [7] is a more general technique to take a snapshot of a set of processes and restore them when it is necessary. Many studies have been conducted to utilize C/R for fault tolerance of long-running high-performance computing applications [14,15,38,47]. CRIU [11] is a user-level tool for Linux that provides C/R of processes and containers. The original CRIU is slow because the snapshot is stored in the secondary storage. However, VAS-CRIU [60] extended CRIU to store a snapshot in memory, reducing the C/R latency. Both are general-purpose C/R techniques that freeze the entire set of processes and take a snapshot of all the resources and all possible kernel-provided states processes might hold. Recently, Groundhog [4] has proposed C/R specialized to serverless computing. As our evaluation results have shown, the memory and performance cost of C/R-based approaches are high as compared to REWIND since REWIND's snapshot feature is supported by the OS kernel and REWIND exploits the temporal locality of function execution to accelerate the next function execution.

6 Conclusions

The lack of temporal isolation exposes modern serverless computing to potential data leakage or exfiltration attacks. The reuse of warm function sandboxes is effective in improving the function execution performance. However, this comes at the risk of allowing persistence points in a sandbox that can be exploited by attackers. The main cause of this problem is the lack of temporal isolation within a shared reused sandbox.

In this paper, we propose REWIND to address the lack of temporal isolation. REWIND rewinds the state of a function sandbox back to its pristine state whenever it completes handling a function request. This allows us to eliminate the persistence points mistakenly produced by the use of a warm sandbox. This eliminates the possibility of leaving privacy-sensitive data in a sandbox. Consequently, REWIND provides the temporal isolation between consecutive function requests. Our evaluation results demonstrated that REWIND provides the temporal isolation at a low cost as compared to the state-of-the-art approach.

Acknowledgments

We would like to thank the anonymous reviewers and shepherd, for their valuable comments. This work was supported in part by Institute for Information & communications Technology Promotion (IITP) grant and by the National Research Foundation of Korea (NRF) grant all funded by the Korea government (MSIT) (No.2021-000773, RS-2023-00321688). Jinkyu Jeong is the corresponding author.

Availability

The source code of the REWIND prototype is available at https://github.com/s3yonsei/rewind_serverless.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 419–434, 2020.

[2] Adil Ahmad, Juhee Kim, Jaebaek Seo, Insik Shin, Pedro Fonseca, and Byoungyoung Lee. Chancel: Efficient multi-client isolation under adversarial programs. In *Network and Distributed Systems Security Symposium (NDSS)*, 2021.

[3] Fritz Alder, N Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. S-faas: Trustworthy and accountable function-as-a-service using intel sgx. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 185–199, 2019.

[4] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. Groundhog: Efficient request isolation in faas. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 398–415, 2023.

[5] Lixiang Ao, George Porter, and Geoffrey M Voelker. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 730–746, 2022.

[6] Lambda execution environments. https://docs.aws.amazon.com/lambda/latest/operatorguide/execution-environments.html, 2023.

[7] Nicholas S. Bowen and Dhiraj K Pradham. Processor- and memory-based checkpoint and rollback recovery. *Computer*, 26(2):22–31, 1993.

[8] Stefan Brenner and Rüdiger Kapitza. Trust more, serverless. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 33–43, 2019.

[9] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, pages 64–78, 2021.

[10] Some upcoming memory-management patches. https://lwn.net/Articles/875970/, 2024.

[11] CRIU. https://criu.org/, 2023.

[12] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. Valve: Securing function workflows on serverless computing platforms. In *Proceedings of The Web Conference 2020*, pages 939–950, 2020.

[13] Pubali Datta, Isaac Polinsky, Muhammad Adil Inam, Adam Bates, and William Enck. {ALASTOR}: Reconstructing the provenance of serverless intrusions. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2443–2460, 2022.

[14] Sheng Di, Mohamed Slim Bouguerra, Leonardo Bautista-Gomez, and Franck Cappello. Optimization of multi-level checkpoint model for large scale hpc applications. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1181–1190. IEEE, 2014.

[15] Sheng Di, Yves Robert, Frédéric Vivien, Derrick Kondo, Cho-Li Wang, and Franck Cappello. Optimization of cloud task processing with checkpoint-restart mechanism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013.

[16] Docker. https://www.docker.com/, 2023.

[17] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, 2020.

[18] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 45–59, 2020.

[19] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 386–400, 2021.

[20] git. https://git-scm.com/, 2023.

[21] Groundhog project repository. https://gitlab.mpi-sws.org/groundhog/, 2023.

[22] Yi He, Roland Guo, Yunlong Xing, Xijia Che, Kun Sun, Zhuotao Liu, Ke Xu, and Qi Li. Cross container attacks: The bewildered {eBPF} on clouds. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5971–5988, 2023.

[23] Benedict Herzog, Stefan Reif, Julian Preis, Wolfgang Schröder-Preikschat, and Timo Hönig. The price of

meltdown and spectre: Energy overhead of mitigations at operating system level. In *Proceedings of the 14th European Workshop on Systems Security*, pages 8–14, 2021.

[24] Benedict Herzog, Stefan Reif, Julian Preis, Wolfgang Schröder-Preikschat, and Timo Hönig. The price of meltdown and spectre: Energy overhead of mitigations at operating system level. In *Proceedings of the 14th European Workshop on Systems Security*, pages 8–14, 2021.

[25] Suman Jana, Donald E Porter, and Vitaly Shmatikov. Txbox: Building secure, efficient sandboxes with system transactions. In *2011 IEEE Symposium on Security and Privacy*, pages 329–344. IEEE, 2011.

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

[27] R. Jones. Gone in 60 Milliseconds: Intrusion and Exfiltration in Server-less Architectures. https://media.ccc.de/v/33c3-7865-gone_in_60_milliseconds, 2023.

[28] Junbin Kang, Le Cai, Feifei Li, Xingxuan Zhou, Wei Cao, Songlu Cai, and Daming Shao. Remus: Efficient live migration for distributed databases with snapshot isolation. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2232–2245, 2022.

[29] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504. IEEE, 2019.

[30] Page Table Isolation (PTI) The Linux Kernel documentation — kernel.org. https://www.kernel.org/doc/html/next/x86/pti.html.

[31] AWS Lambda. Aws lambda custmoer case studies. https://aws.amazon.com/lambda/resources/customer-case-studies/, 2020.

[32] Eliezer Levy and Abraham Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys (CSUR)*, 22(4):321–374, 1990.

[33] Ping-Min Lin and Alex Glikson. Mitigating cold starts in serverless platforms: A pool-based approach. *arXiv preprint arXiv:1903.12221*, 2019.

[34] Linux kernel. https://www.kernel.org/, 2023.

[35] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 218–233, 2017.

[36] Osman Surkatty Mayank Thakkar, Marc Brooker. Security overview of aws lambda. Technical report, AWS, 2022.

[37] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[38] Bogdan Nicolae and Franck Cappello. Blobcr: Efficient checkpoint-restart for hpc applications on iaas clouds using virtual disk image snapshots. In *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2011.

[39] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. {SOCK}: Rapid task provisioning with {Serverless-Optimized} containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, 2018.

[40] Ocfs2. https://ocfs2.wiki.kernel.org/, 2015.

[41] Kaan Onarlioglu, Collin Mulliner, William Robertson, and Engin Kirda. Privexec: Private execution as an operating system service. In *2013 IEEE Symposium on Security and Privacy*, pages 206–220. IEEE, 2013.

[42] Apache OpenWhisk. https://openwhisk.apache.org/, 2023.

[43] OverlayFS. https://kernel.org/doc/html/latest/filesystems/overlayfs.html, 2023.

[44] Linux kernel document - overlayfs. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt, 2023.

[45] Apache openWhisk Runtimes for Docker. https://github.com/apache/openwhisk-runtime-docker, 2023.

[46] Apache openWhisk Runtimes for Python. https://github.com/apache/openwhisk-runtime-python, 2023.

[47] Konstantinos Parasyris, Kai Keller, Leonardo Bautista-Gomez, and Osman Unsal. Checkpoint restart support for heterogeneous hpc applications. In *2020 20th*

IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), pages 242–251. IEEE, 2020.

[48] Donald E Porter, Owen S Hofmann, Christopher J Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 161–176, 2009.

[49] Weizhong Qiang, Zezhao Dong, and Hai Jin. Se-lambda: Securing privacy-sensitive serverless applications using sgx enclave. In *International Conference on Security and Privacy in Communication Systems*, pages 451–470. Springer, 2018.

[50] Xiang Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An analysis of performance evolution of linux's core operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 554–569, 2019.

[51] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[52] Jay Shah and Dushyant Dubaria. Building modern clouds: using docker, kubernetes & google cloud platform. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0184–0189. IEEE, 2019.

[53] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, 2020.

[54] Livio Soares and Michael Stumm. {FlexSC}: Flexible system call scheduling with {Exception-Less} system calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[55] Soft-Dirty PTEs — docs.kernel.org. https://docs.kernel.org/admin-guide/mm/soft-dirty.html, 2013.

[56] Hillel Sollow. Top 4 reasons why serverless is secure. https://blog.checkpoint.com/2020/07/13/top-4-reasons-why-serverless-is-secure, 2022.

[57] Noah Spahn, Nils Hanke, Thorsten Holz, Christopher Kruegel, and Giovanni Vigna. Container orchestration honeypot: Observing attacks in the wild. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 381–396, 2023.

[58] Amoghavarsha Suresh, Gagan Somashekar, Anandh Varadarajan, Veerendra Ramesh Kakarla, Hima Upadhyay, and Anshul Gandhi. Ensure: Efficient scheduling and autonomous resource management in serverless environments. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 1–10. IEEE, 2020.

[59] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 559–572, 2021.

[60] Ranjan Sarpangala Venkatesh, Till Smejkal, Dejan S Milojicic, and Ada Gavrilovska. Fast in-memory criu for docker containers. In *Proceedings of the International Symposium on Memory Systems*, pages 53–65, 2019.

[61] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast {RDMA-codesigned} remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 497–517, 2023.

[62] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 30–44, 2020.

[63] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 724–739, 2021.

[64] Shixuan Zhao, Pinshen Xu, Guoxing Chen, Mengya Zhang, Yinqian Zhang, and Zhiqiang Lin. Reusable enclaves for confidential serverless computing. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4015–4032, 2023.

[65] Chao Zheng, Lukas Rupprecht, Vasily Tarasov, Douglas Thain, Mohamed Mohamed, Dimitrios Skourtis, Amit S Warke, and Dean Hildebrand. Wharf: Sharing docker images in a distributed file system. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 174–185, 2018.

A Artifact Appendix

A.1 Abstract

To reproduce the experimental results of REWIND presented in Section 4 of the paper, we provide the source code and scripts. The source code includes the Linux kernel code with REWIND's snapshot creation and rewind operations as described in the paper. The scripts contain the execution code for validating the experimental results shown in Figures 5-13. By following the provided instructions, users can expect to obtain results similar to those presented in the paper.

A.2 Scope

This artifact includes the kernel and Python code implementing REWIND's functionality, as well as Docker containers with REWIND's snapshot creation and rewind capabilities. Additionally, the artifact provides workload codes and scripts for evaluating REWIND. The workloads and scripts provided in the artifact allow for the reproduction of the results presented in the paper.

A.3 Contents

The provided `README.md` describes the artifact and offers guidelines for evaluation. Furthermore, the `README.md` details the directory structure of the REWIND repository.

A.4 Hosting

The artifact can be downloaded from the main branch on GitHub at https://github.com/s3yonsei/rewind_serverless

A.5 Requirements

A.5.1 Hardware requirement

To run the artifact, REWIND requires an Intel CPU. While our experiments utilized the Intel Xeon Gold 5118, any Intel architecture CPU is compatible with REWIND.

A.5.2 Software requirement

The experiments provided in this artifact are designed to run within the OpenWhisk and Docker container environment. To reproduce the results, we recommend using the OpenWhisk and Docker container images provided with the artifact.