Towards Efficient End-to-End Encryption for Container Checkpointing Systems

Radostin Stoyanov*
University of Oxford
Oxford, United Kingdom
radostin.stoyanov@eng.ox.ac.uk

Adrian Reber
Red Hat
Stuttgart, Germany
areber@redhat.com

Daiki Ueno
Red Hat
Tokyo, Japan
dueno@redhat.com

Michał Cłapiński
Google
Warsaw, Poland
mclapinski@google.com

Andrei Vagin
Google
Seattle, Washington, USA
avagin@google.com

Rodrigo Bruno
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal
rodrigo.bruno@tecnico.ulisboa.pt

ABSTRACT

Container checkpointing has emerged as a new paradigm for task migration, preemptive scheduling and elastic scaling of microservices. However, as soon as a snapshot that contains raw memory is exposed through the network or shared storage, sensitive data such as keys and passwords may become compromised. Existing solutions rely on encryption to protect data included in snapshots but by doing so prevent important performance optimizations such as memory de-duplication and incremental checkpointing. To address these challenges, we design and implement CRIUsec, an efficient end-to-end encryption scheme for container checkpointing systems built on the open-source CRIU (Checkpoint/Restore In Userspace). Our preliminary evaluation shows that CRIUsec integrates seamlessly with popular container platforms (Docker, Podman, Kubernetes), and compared to existing solutions, achieves an average of 1.57× speedup for memory-intensive workloads, and can be up to 100× faster for compute-intensive workloads.

CCS CONCEPTS

• **Security and privacy → Virtualization and security**.

*Also with Red Hat.

APSys '24, September 04–05, 2024, Kyoto, Japan

ACM ISBN 979-8-4007-1105-3/24/09.
https://doi.org/10.1145/3678015.3680477

KEYWORDS

CRIU, Containers, Checkpointing, Security

ACM Reference Format:
Radostin Stoyanov, Adrian Reber, Daiki Ueno, Michał Cłapiński, Andrei Vagin, and Rodrigo Bruno. 2024. Towards Efficient End-to-End Encryption for Container Checkpointing Systems. In *15th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '24), September 4–5, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3678015.3680477

1 INTRODUCTION

Over the past decade, container checkpointing has been adopted into many popular container platforms to enable use cases such as fault-tolerance [30], fast application start-up [11, 29], live migration [24], elastic scaling [15], preemptive scheduling [12, 22], and forensic analysis [1]. Despite these benefits, checkpoint/restore tools simply dump memory pages to disk and these pages may include sensitive data such as cryptographic keys, API access tokens, and passwords. The traditional solution to this problem is to use encryption. However, container engines lack native support for checkpoint encryption and require infrastructure operators to implement their own security mechanisms using external encryption tools. These tools lack visibility on what is included in the snapshot and encrypt/decrypt all of the dumped application state. This problem is particularly important in multi-tenant environments, where container platforms are shared across multiple users, teams, or organizations with different security requirements.

Existing security mechanisms for container checkpointing can be classified into two categories based on the method of processing data: (i) *local encryption* – checkpoint data is saved and encrypted locally before being transferred to a remote or distributed storage [2, 3, 7, 20]; and (ii) *streaming*

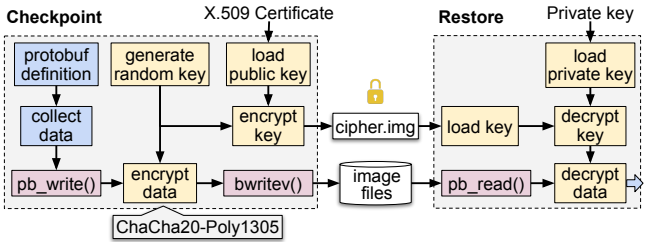


Figure 1: Checkpoint/restore workflow for encrypted CRIU images in protocol-buffer format. Highlighted in yellow are components providing CRIUsec functionality, those in red are modified, while those in blue remain unmodified.

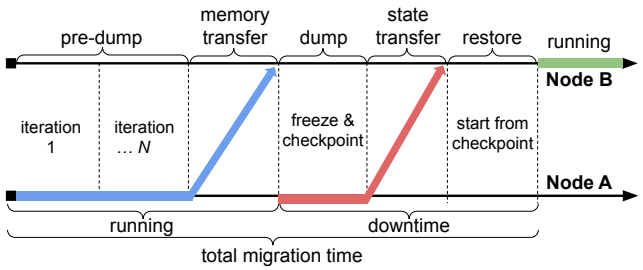


Figure 2: Pre-copy container migration with iterative checkpointing involves transferring the majority of memory state over *N* iterations prior to freezing, transferring the remaining state, and restarting the application from the checkpoint.

encryption – a data streaming pipeline is used to encrypt checkpoint data on-the-fly [13, 23, 28].

While local encryption offers data protection, the required intermediate storage introduces a significant performance overhead, increased storage requirements, and creates a new attack vector (through unauthorized storage access [16, 21]). Streaming, despite offering a more efficient and secure solution, relies on data transfer through undocumented, ad hoc protocol implemented by operators over UNIX domain sockets. This creates a potential vulnerability where an attacker with filesystem access could replace the original socket file and perform a man-in-the-middle attack. This presents implementation challenges that prevent mainstream adoption of streaming encryption in container engines. In addition, neither of these two approaches supports optimization techniques such as incremental checkpointing and memory deduplication. These techniques are crucial for performance and cannot be efficiently implemented with existing external snapshot encryption mechanisms (both local and streaming), as they require multiple rounds of encryption and decryption to calculate the incremental changes and de-duplicate clear text data.

To address these challenges, we propose CRIUsec, an end-to-end encryption mechanism for container checkpointing systems. CRIUsec combines asymmetric cryptography with symmetric stream and block ciphers to provide confidentiality, integrity, and authentication of checkpoint data. It minimizes the overhead of encryption and storage requirements by encrypting the checkpoint data in-place and on-the-fly. In addition, it enables support for incremental checkpointing and memory de-duplication by integrating encryption mechanisms with the core checkpoint/restore functionality.

Our preliminary evaluation demonstrates that compared to state-of-the-art cryptographic tools, CRIUsec significantly reduces the overheard of end-to-end encryption. It achieves an average reduction of up to 62% for memory-intensive and can be up to two orders of magnitude faster for compute-intensive workloads. In addition, CRIUsec seamlessly integrates with popular container platforms (Docker, Podman, Kubernetes) without modifications to existing workflows.

2 BACKGROUND

Container platforms such as Kubernetes typically assume a threat model where the hardware, operating system (OS) kernel, and privileged userspace are trusted. Container runtimes (e.g., runc) are responsible for handling low-level tasks such as creating and running containers, while container engines (e.g., CRI-O) are high-level tools that fetch and manage container images, and act as an intermediary between Kubernetes and the underlying container runtime. Container checkpoint/restore is enabled at low-level using tools such as CRIU [5] to transparently capture the state of a running container and restore it later on the same or different machine.

2.1 Security Risks

Container checkpointing with platforms such as Docker, Podman, Kubernetes by default does not provide support for encryption. System administrators are required to setup their own security mechanisms that restrict access to checkpoint data through methods such as access control lists or filesystem encryption. The lack of a security standard for container checkpoints creates a critical vulnerability that leaves sensitive data unencrypted and exposed to unauthorized access.

2.2 End-to-End Encryption

While alternative methods can be utilized to protect checkpoint data, our aim is to provide a generic solution for *end-to-end encryption*, i.e., checkpoint data is encrypted while it is extracted from the address space of processes, and decrypted as it is restored within the new process address space. By addressing the problem at the low-level checkpoint/restore functionality, it ensures that optimization techniques such as

iterative checkpointing (illustrated in Figure 2) and memory de-duplication remain available while at the same time avoiding expensive decryption and encryption cycles. In addition, this approach integrates seamlessly with existing checkpointing systems.

3 CRIU OVERVIEW AND THREAT MODEL

CRIUsec aims to provide security by integrating cryptographic capabilities directly into the checkpointing engine (CRIU). This approach ensures that sensitive data is automatically encrypted end-to-end, and eliminates the need for external data processing layers or tools. In addition, it applies fine-grained encryption of checkpoint data where individual memory pages and data structures can be modified without the need to decrypt the entire checkpoint. This is crucial for enabling advanced optimizations such as iterative checkpointing and memory de-duplication.

To optimize performance, CRIUsec utilizes in-place encryption and zero-copy techniques, thus avoiding unnecessary data copying. This method is particularly important for memory-intensive workloads, where the overhead of encryption can significantly increase downtime during checkpointing.

3.1 System Model

A container checkpoint consists a set of files (CRIU images) that capture the serialized runtime state of a process tree, including open file descriptors, user/group IDs, PIPEs, sockets, mount points, and memory state. Understanding the characteristics of different types of CRIU images (such as the number of data entries and size) is crucial for selecting the most suitable encryption cipher. In particular, CRIU images can be broadly classified into the following three categories:

- **Memory pages:** binary data with large number of entries of fixed size;
- **Raw images:** data in a third-party format defined by external tools such as `ip`, `iptables`, `tar`;
- **Protobuf images:** structured data in protocol buffers format with a variable number of entries and size [26].

For each category, CRIUsec provides a separate component encapsulating the cryptographic operations for encryption, decryption, and verification. These components are reused across all images of a specific data format, aiming to avoid code duplication and improve maintainability. We discuss in more detail the implementation in Section 4.

3.2 Threat Model and Assumptions

Our threat model is primarily concerned with vulnerabilities that may be exploited to compromise the confidentiality or integrity of checkpoint data. Attacks in scope include compromising the host environment (e.g., through container breakout [17]) that allows an attacker to read/modify checkpoints stored on persistent storage or to intercept network communications. We assume that a container does not voluntarily reveal its own private data whether on purpose or by accident, but attacks with malicious containers where the attacker has control over the container state being checkpointed are in scope. Availability, physical or side-channel attacks are out of scope.

We assume secure key and certificate storage is available, with certificates signed by a trusted Certificate Authority. We assume the hardware is bug-free, and the system is initially benign, allowing signatures and keys to be securely stored before the system is compromised. We assume the container engine and runtime do not have any vulnerabilities and can be trusted. We assume it is infeasible to perform brute-force attacks, and encryption protocols are designed to defend against replay attacks.

4 CRIUSEC

CRIUsec builds on the open-source CRIU [5] and leverages cryptographic primitives from the GnuTLS library [23] to apply the most appropriate cipher for each CRIU image type. The implementation consists of approximately 2,000 lines of C code, and 250 lines of Python code extending the CRIU image tool (CRIT) and the zero downtime migration (ZDTM) test suite.

4.1 Encrypting Protocol Buffer Entries

Checkpoint data in protobuf format is stored as *single-entry* or *array* data structure. A single-entry format stores a protobuf message (entry) with pre-defined fields (e.g., process names, signal masks, registers). In contrast, array image has zero or more entries, and this data format is utilized for the majority of checkpoint data (e.g., process tree, files, sockets).

To ensure security and efficient processing for these messages, CRIUsec applies ChaCha20-Poly1305 authenticated encryption with associated data. The ciphertext size is then prepended, and authentication tag with random nonce (number used only once) are appended to the data. During restore, an analogous approach is used for decryption.

Since, during restore, the content of array images is read until an end-of-file (EOF) has been reached, the authentication tag, nonce, and ciphertext size are prepended to the encrypted data for each entry in the array. This approach allows for detecting data corruption, e.g., when EOF is encountered unexpectedly. Figure 1 illustrates this checkpoint/restore workflow, where `pb_read` and `pb_write` are utilized to read and write protobuf entries, `bwritev` combines multiple data streams with vectored I/O and writes to persistent storage.

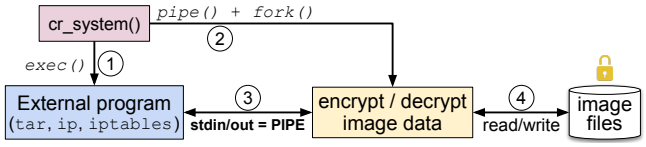


Figure 3: Pipeline for encrypting/decrypting raw images created with external tools. The external tool is executed with `cr_system` ①, while a helper function handles read/write operations to the image file ②, and intercepts the input/output data ③ to perform on-the-fly encryption and decryption ④.

4.2 Encrypting Raw Images

When CRIU collects and restores data with tools such as `ip`, `iptables`, and `tar`, these tools are executed with standard I/O file descriptors set to transfer data between the tools and a checkpoint (raw) image. CRIUsec replaces the I/O file descriptors with PIPEs that allow intercepting the checkpoint data to perform encryption and decryption on-the-fly. This pipeline mechanism is integrated with the so-called `cr_system` component (as shown in Figure 3). This component is responsible for running the external tools, handling the `SIGCHLD` signal and exit code.

4.3 Encrypting Memory Pages

Memory pages typically make up the largest part of a checkpoint [27]. The techniques for checkpoint/restore of memory are highly optimized for performance and efficiency, leveraging zero-copy transfer between processes and file descriptors to avoid unnecessary data copying (e.g., using `splice`, `vmsplice`, `process_vm_readv`). This memory state consists of three types of images: *memory mappings* (mm), *memory page mappings* (pagemap), and *memory pages* (pages). The first two image types are encoded with protobuf format, and encryption is handles as described above.

The third image type consists of individual pages stored sequentially in binary format (with 4 KB size on x86-64 systems) according to the mappings in the pagemap image. CRIUsec applies encryption with the AES-XTS block cipher with a single initialization vector (IV). This approach allows to avoid saving authentication tag and nonce for each memory page and to minimize storage overhead. It is important to note that IV is set with random generated data that is unique for each checkpoint and it is never reused. This method provides confidentiality and ensure that encryption of identical plaintext will not result in the same ciphertext.

To enable performance optimizations such as asynchronous restore of memory [8] and de-duplication, each page is encrypted independently. The implementation of this approach is straightforward during checkpointing, however,

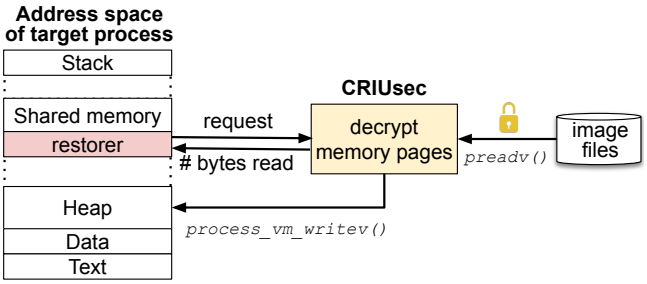


Figure 4: Restoring encrypted memory pages within PIE-compiled restore context with a helper process decrypting the data.

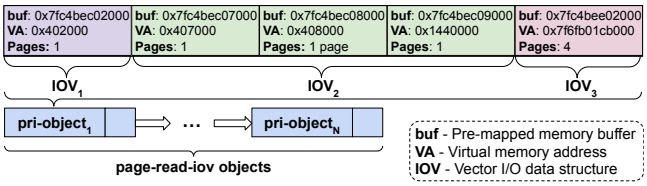


Figure 5: The page-read-iov data structure is used to queue asynchronous *read* requests during the restore operation. This helps to reduce the number of system calls used for reading data and restoring memory mappings. It consists of a linked list with objects, where each object contains one or more vectors, and each vector may contain pages with different virtual addresses.

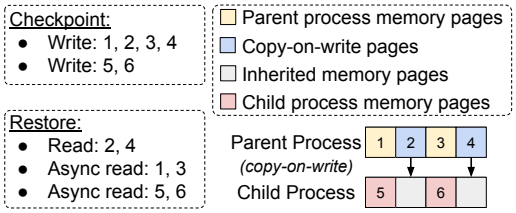


Figure 6: An illustrative example of saving and restoring memory pages for parent and child process.

the restore operation is performed in PIE-compiled code (*restorer context*) that cannot be linked with GnuTLS to perform cryptographic operations. To address this problem, we introduce a helper process that communicates with the PIE code via PIPEs, performs decryption, and writes the memory pages at the target address space using `process_vm_writev` system call, as shown in Figure 4.

4.3.1 Integrity Verification. While AES-XTS provides efficient security solution, it requires an additional mechanism for integrity verification. A traditional solution for this problem is to compute a keyed-hash message authentication code (HMAC) for the encrypted data. However, the order in which

pages are decrypted during restore is different from the order in which they are encrypted during checkpointing. Figures 5 and 6 show an example of how asynchronous read requests are combined during restore with a single invocation of the `preadv` system call. In particular, during restore data is read into multiple scatter input (IOV) buffers that are then remapped to their original virtual address. CRIUsec uses an approach inspired by Merkle trees that verifies the integrity of memory pages by computing an HMAC digest of the virtual address, process ID, and ciphertext of each memory page. It then computes an XOR for all HMAC digest values and stores the final result in the checkpoint. The restore process computes HMAC digest values in a similar manner and compares the final result with the digest in the checkpoint, and failing with an error in case of a mismatch.

4.4 Iterative Checkpointing

CRIU supports two core functionalities that enable iterative checkpointing: (i) a memory tracking mechanism that identifies modified (dirty) pages; and (ii) a *pre-dump* feature that creates a memory snapshot of only the changed pages since the last checkpoint. The memory tracking functionality can be used as a fault-tolerance mechanism. This can be achieved by periodically creating a full snapshot and storing only modified memory pages. Alternatively, it can be utilized in combination with pre-dump during live-migration. This approach reduces downtime by minimizing the amount of data that needs to be transferred between source and destination nodes when the target application is not running. CRIUsec integrates cryptographic capabilities with this mechanism that allow to decrypt only the necessary data to identify pages present in previous checkpoints.

4.5 Memory De-duplication

Memory de-duplication can be used both during checkpointing to remove redundant data in previous checkpoints, and during restore to *punch holes* (with `fallocate` syscall) in the checkpoint image file as soon as memory pages are restored. CRIUsec encrypts memory pages using in-place encryption that guarantees encrypted memory blocks to have the same size as the plantext data. This allows to de-duplicate memory without the need to decrypt the content of individual pages.

5 PRELIMINARY EVALUATION

We evaluate the performance of CRIUsec compared to baseline CRIU, which does not provide support for encryption, as well as CRIU when used in conjunction with general-purpose encryption tools. In particular, we address the following questions in this section:

- What is the performance overhead of encryption during checkpointing?
- How does the performance of CRIUsec compare to other state-of-the-art encryption tools?
- What is the impact of encryption on application downtime during checkpointing?

5.1 Experimental Setup and Methodology

We run all experiments on Ubuntu 22.04 server with kernel v5.15 running on Intel i9 CPU (3.50 GHz, 8 cores, 16 threads), Corsair 128 GB DDR4 memory (2133 MHz), and 1 TB Samsung SSD 970 NVMe disk. We use runc v1.1.12, Podman v4.5, Docker v26.0.1, Kubernetes v1.27.4, and build CRIUsec on top of CRIU v3.19 compiled with GnuTLS v3.8.5.

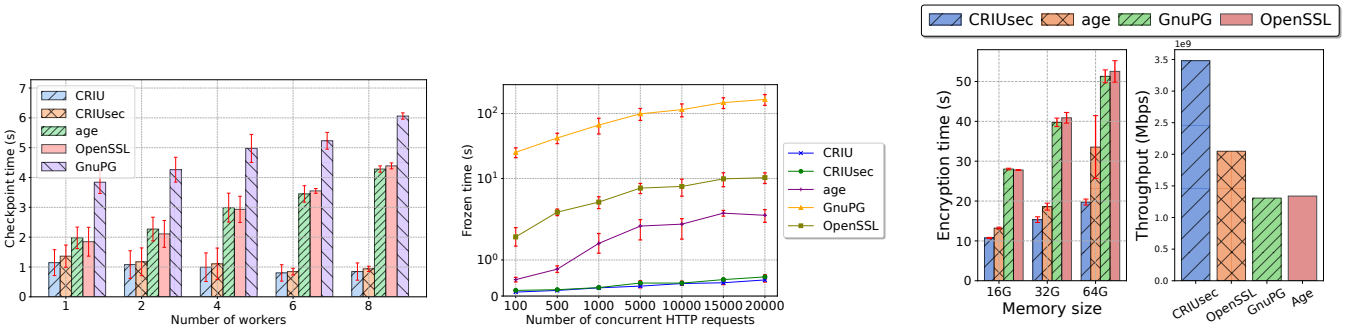
We measure the performance of CRIUsec with and without encryption, and use OpenSSL v3.0.2 [6], GnuPG v2.2.27 [19], and age v1.1.1 [25] as baselines. These tools run via CRIU's *action-script* functionality, as a shell script that implements the *post-dump* hook to encrypt all CRIU image files at the final phase of the checkpointing process.

Our evaluation focuses on encryption with two types of workloads: *memory-intensive* and *compute-intensive*. Memory-intensive workloads have a large memory state, and a checkpoint typically consists of a small number of very large files (that contain memory pages). In contrast, compute-intensive workloads run a large numbers of processes or threads, and checkpointing generates many small files (corresponding to the process tree).

Each experiment is repeated 10 times and standard deviation is calculated to quantify uncertainty of measurements. To ensure consistency and accuracy of the performance measurements, we disable CPU frequency scaling, unnecessary systemd services, and use a real-time scheduling policy with high priority.

5.2 Performance Overhead

To evaluate the encryption overhead of CRIUsec during checkpointing, we utilize CRIU's built-in *stats* functionality [18], extending it to measure encryption and verification times for steam and block ciphers. For comparison, we measure the total execution time of the action-script providing similar encryption functionality with OpenSSL using AES-256-CBC, GnuPG using AES-256 cipher algorithm without compression, and age with ChaCha20-Poly1305. Figure 7 shows the results of checkpointing a container running stress-ng, an HTTP server, and a memhog process with different *size* parameters. The results in Figure 7 (a) demonstrate that increasing the number of workers for a performance benchmark running across different CPUs has only a minimal impact on the encryption overhead of CRIUsec, and it is up to 2.82× faster when compared to encryption with



(a) Checkpointing time of Podman container with runtime (runc) for compute-intensive workload (stress-ng) with increasing number of workers.

(b) Frozen time (in seconds) during checkpointing of a Podman container running Apache HTTP server processing concurrent requests generated with ApacheBench [10].

(c) Comparison of encryption time and throughput with different encryption methods for a memory-intensive workload (memhog).

Figure 7: Performance comparison of CRIUsec with OpenSSL, GnuPG, and age for compute-intensive workload, containerized HTTP server, and memory-intensive workload.

external tools. Similarly, the results in Figure 7 (b) show that when checkpointing a container running a real-world application with a large number of concurrent threads, CRIUsec is up to 100× faster than the slowest option (GnuPG). Figure 7 (c) highlights throughput improvements for memory-intensive workloads, with CRIUsec achieving up to 2.6× faster encryption.

5.3 Application downtime

For the purposes of our evaluation, we define *downtime* to be the time during checkpointing when the target application is in a *frozen* state. Although encrypting the data after a checkpoint has been created does not have an impact on the frozen time, end-to-end encryption requires cryptographic computations to be performed during checkpointing process, and this increased security comes at the cost of potentially longer downtime. Figure 7 (b) shows that the frozen times during checkpointing with CRIUsec are significantly lower compared to alternative solutions.

6 DISCUSSION

While CRIUsec aims to minimize the performance overhead of encryption by integrating cryptographic primitives with the checkpoint/restore functionality, it inevitably introduces some computational overhead relative to unencrypted checkpointing. Compared to alternative solutions that rely on general-purpose encryption tools (e.g., OpenSSL, GnuPG, age), CRIUsec achieves an improved trade-off between security and performance. However, this approach is specific to the data format used by CRIU. Other checkpointing systems such as DMTCP [4] or systems that offer checkpointing functionality like PyTorch [14] and Apache Flink [9] may require different solutions to achieve efficient end-to-end encryption. As CRIU has been integrated with a wide range of container platforms (e.g., Docker, Podman, Kubernetes, OpenVz, LXC/LXD), CRIUsec can be seamlessly integrated with existing container checkpointing systems to enable checkpoint encryption.

7 CONCLUSION AND FUTURE WORK

In this paper, we propose CRIUsec – an efficient, end-to-end encryption scheme for container checkpointing systems. It provides built-in encryption optimized to apply an appropriate cipher based on the data format of each checkpoint image, and improves performance and storage efficiency with zero-copy techniques and in-place encryption. We evaluated CRIUsec using memory-intensive and compute-intensive workloads, and compared its performance against state-of-the-art cryptographic tools: OpenSSL, GnuPG, and age. Our evaluation results show that end-to-end encryption with CRIUsec outperforms alternative solutions by up to two orders of magnitude, resulting in significantly reduced storage utilization and application downtime. Future work will focus on integrating CRIUsec with data compression techniques to further optimize storage efficiency.

ACKNOWLEDGMENTS

We would like to express our gratitude to the anonymous reviewers for their insightful comments. We also extend our thanks to Wes Armour for his invaluable contributions to system design, and to Hubert Kario for their helpful feedback on the implementation. This work is partly supported by the EPSRC Doctoral Training Partnership (DTP) [grant number EP/T517811/1].

REFERENCES

[1] Adrian Reber. 2023. Forensic container checkpointing in Kubernetes. https://kubernetes.io/blog/2022/12/05/forensic-container-checkpointing-alpha/ Accessed: 2023-08-15.

[2] Adrian Reber. 2023. Kubelet Checkpoint API. https://kubernetes.io/docs/reference/node/kubelet-checkpoint-api/ Accessed: 2023-08-15.

[3] Amazon. 2024. Use checkpoints in Amazon SageMaker. https://docs.aws.amazon.com/sagemaker/latest/dg/model-checkpoints.html

[4] Jason Ansel, Kapil Arya, and Gene Cooperman. 2009. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. 1–12. https://doi.org/10.1109/IPDPS.2009.5161063

[5] CRIU. 2024. Checkpoint/Restore In Userspace. https://criu.org/ Accessed: 2024-03-21.

[6] OpenSSL Project Developers. 2024. *OpenSSL*. OpenSSL Project. https://www.openssl.org/

[7] Docker. 2024. Docker Checkpoint. https://docs.docker.com/reference/cli/docker/checkpoint/

[8] Pavel Emelyanov. 2016. Implement asynchronous mode of reading pages. https://github.com/checkpoint-restore/criu/commit/8f2b2c7f0baa491331bdcde915d3278bf624d459.

[9] Apache Software Foundation. 2024. Apache Flink Checkpoints. https://nightlies.apache.org/flink/flink-docs-master/docs/ops/state/checkpoints/. Accessed: 2024-07-14.

[10] The Apache Software Foundation. 2024. ApacheBench - A Benchmarking Tool for HTTP Servers. https://httpd.apache.org/docs/2.4/programs/ab.html Accessed: 21-04-2024.

[11] Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, Joao Carreira, and Pedro Fonseca. 2024. Pronghorn: Effective Checkpoint Orchestration for Serverless Hot-Starts. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 298–316. https://doi.org/10.1145/3627703.3629556

[12] Jack Li, Calton Pu, Yuan Chen, Vanish Talwar, and Dejan Milojicic. 2015. Improving Preemptive Scheduling with Application-Transparent Checkpointing in Shared Clusters. In *Proceedings of the 16th Annual Middleware Conference* (Vancouver, BC, Canada) *(Middleware '15)*. Association for Computing Machinery, New York, NY, USA, 222–234. https://doi.org/10.1145/2814576.2814807

[13] Victor Marmol and Andy Tucker. 2018. Task Migration at Scale Using CRIU. Linux Plumbers Conference. https://lpc.events/event/2/contributions/69/ Accessed: 2024-01-30.

[14] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. 2021. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, USA, 203–216. https://www.usenix.org/conference/fast21/presentation/mohan

[15] Ritesh Naik. 2021. Container Checkpoint/Restore at Scale for Fast Pod Startup Time. https://kccncna2021.sched.com/event/lV0Y/container-checkpointrestore-at-scale-for-fast-pod-startup-time-ritesh-naik-mathworks

[16] Hyochang Nam, Jong Kim, Sung Je Hong, and Sunggu Lee. 2003. Secure checkpointing. *Journal of Systems Architecture* 48, 8 (2003), 237–254. https://doi.org/10.1016/S1383-7621(02)00137-6

[17] National Vulnerability Database (NVD). 2024. CVE-2024-21626. https://github.com/advisories/GHSA-xr7r-f8xq-vfvv GitHub repository.

[18] CRIU Project. 2019. Statistics. https://criu.org/Statistics Accessed: 2024-04-23.

[19] GNU Project. 2024. *GNU Privacy Guard (GnuPG)*. GnuPG Project. https://www.gnupg.org/

[20] Adrian Reber. 2024. Podman Checkpoint. https://docs.podman.io/en/latest/markdown/podman-container-checkpoint.1.html

[21] Prakhar Sah and Matthew Hicks. 2023. Hitchhiker's Guide to Secure Checkpointing on Energy-Harvesting Systems. In *Proceedings of the 11th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems (ENSsys '23)*. Association for Computing Machinery, New York, NY, USA, 8–15. https://doi.org/10.1145/3628353.3628542

[22] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, Nipun Kwatra, Ramachandran Ramjee, Pankaj Sharma, Atul Katiyar, Vipul Modi, Vaibhav Sharma, Abhishek Singh, Shreshth Singhal, Kaustubh Welankar, Lu Xun, Ravi Anupindi, Karthik Elangovan, Hasibur Rahman, Zhou Lin, Rahul Seetharaman, Cheng Xu, Eddie Ailijiang, Suresh Krishnappa, and Mark Russinovich. 2022. Singularity: Planet-Scale, Preemptive and Elastic Scheduling of AI Workloads. arXiv:2202.07848 [cs.DC]

[23] Radostin Stoyanov. 2019. Secure Image-less Container Migration. Linux Plumbers Conference. https://lpc.events/event/4/contributions/478/ Accessed: 2023-01-30.

[24] Radostin Stoyanov and Martin J. Kollingbaum. 2018. Efficient Live Migration of Linux Containers. In *High Performance Computing*, Rio Yokota, Michèle Weiland, John Shalf, and Sadaf Alam (Eds.). Springer International Publishing, Cham, 184–193.

[25] Filippo Valsorda. 2024. *age: A Simple, Modern and Secure File Encryption Tool*. Age Encryption. https://github.com/FiloSottile/age

[26] Kenton Varda. 2008. *Protocol Buffers: Google's Data Interchange Format*. Technical Report. Google. https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html

[27] Ranjan Sarpangala Venkatesh, Till Smejkal, Dejan S. Milojicic, and Ada Gavrilovska. 2019. Fast in-memory CRIU for docker containers. In *Proceedings of the International Symposium on Memory Systems* (Washington, District of Columbia, USA) *(MEMSYS '19)*. Association for Computing Machinery, New York, NY, USA, 53–65. https://doi.org/10.1145/3357526.3357542

[28] Nicolas Viennot. 2020. Fast checkpointing with criu-image-streamer. Linux Plumbers Conference. https://lpc.events/event/7/contributions/641/ Accessed: 2023-01-30.

[29] 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

[30] Diyu Zhou and Yuval Tamir. 2020. Fault-Tolerant Containers Using NiLiCon. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Institute of Electrical and Electronics Engineers (IEEE), New Orleans, LA, USA, 1082–1091. https://doi.org/10.1109/IPDPS47924.2020.00114