# Secure HPC: A Workflow Providing a Secure Partition on an HPC System

Hendrik Nolte[a,*], Nicolai Spicher[b], Andrew Russel[c], Tim Ehlers[a], Sebastian Krey[a], Dagmar Krefting[b] and Julian Kunkel[a]

[a]*Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen, Germany*
[b]*Institut für Medizinische Informatik der Universitätsmedizin Göttingen, Germany*
[c]*Cornelis Networks Inc, USA*

ARTICLE INFO

*Keywords*:
high performance computing
sensitive data
secure computing
data encryption

ABSTRACT

Driven by the progress of data and compute-intensive methods in various scientific domains, there is an increasing demand from researchers working with highly sensitive data to have access to the necessary computational resources to be able to adapt those methods in their respective fields. To satisfy the computing needs of those researchers cost-effectively, it is an open quest to integrate reliable security measures on existing High Performance Computing (HPC) clusters. The fundamental problem with securely working with sensitive data is, that HPC systems are shared systems that are typically trimmed for the highest performance – not for high security. For instance, there are commonly no additional virtualization techniques employed, thus, users typically have access to the host operating system. Since new vulnerabilities are being continuously discovered, solely relying on the traditional Unix permissions is not secure enough. In this paper, we discuss *Secure HPC*, a workflow allowing users to transfer, store and analyze data with the highest privacy requirements. Our contributions are the design of a multi-node secure workflow with parallel I/O, a strict security model enforced by the system and network features, and lastly the demonstration of a medical use case. In our experiments, we see an advantage in the asynchronous execution of IO requests in `dm_crypt`, while reaching 80% of the ideal performance. When comparing `eCryptFS` with `GoCryptFS` as two representative filesystem-level encryption stacks, eCryptFS was twice as fast. In a real use case, we observed on average 97% of the native performance.

## 1. Introduction

The increasing adoption of data and compute-intensive algorithms in digital humanities or life sciences (Uecker, Ong, Tamir, Bahri, Virtue, Cheng, Zhang and Lustig, 2015; Hammernik, Klatzer, Kobler, Recht, Sodickson, Pock and Knoll, 2018) has drastically increased the demand for cost-effective solutions in research domains that are subjected to very strict data security restrictions, like the General Data Protection Regulation (GDPR) or the Health Insurance Portability and Accountability Act (HIPAA). Historically, HPC systems in public data centers serve those tasks for insensitive data for capacity as well as capability computing. Here, different users share the available resources and can run their compute jobs simultaneously on shared or exclusive subsets of nodes. Due to the optimization for performance, it is very common, that users interact directly with the operating system of the host. Users are trusted to some extent, and, thus, any local vulnerability can be immediately exploited by users or bots that gained control of user credentials. Taking into account that there are continuously new attacks discovered that lack a reliable solution over a sustained period of time (Jattke, van der Veen, Frigo, Gunter and Razavi), sensitive data should only be transferred, stored, and processed with care in public data centers. Some industry and government data centers (such as for weapon research) limit access and employ strict policies regarding system access, even to the point where sensitive data is physically disconnected if not needed. However, restricting system access does not resolve the problem with the data access, since administrators basically have full access. We believe, that even in the case of a privilege escalation leading to a compromised cluster, the integrity of data should be guaranteed.

This article extends our previous work (Nolte, Sarmiento, Ehlers and Kunkel, 2022), where a generic workflow to transfer, store and process sensitive data on a shared HPC system is presented. The previous workflow is limited to single nodes. In this paper, the original idea was extended to support multi-node applications while providing an even higher security standard compared to the single node setup. The higher security standard is achieved by the higher degree of network isolation of the compute nodes which now also involves network partitioning of the high speed interconnect. There are several challenges involved when scaling from a single node to multi node support. The first challenge is to ensure a secure communication, which can be done either by encrypting and decrypting the messages on the nodes, or by partitioning the entire network. The second challenge is to support as many native multi-node application as possible, which partly rely on parallel I/O across multiple nudes, which, however, needs to be encrypted locally on each node to guarantee that no unencrypted data is written to the global filesystem. The third challenge is to guarantee proper user authentication, to ensure that even the `root` user has no access

*Corresponding author
hendrik.nolte@gwdg.de (H. Nolte)
ORCID(s): 0000-0003-2138-8510 (H. Nolte)

to the nodes and therefor to the unencrypted data, and proper node initialization to ensure that the necessary decryption keys are available on all nodes. Our new contributions are:

- extension to multi-node secure workflows
- support for parallel I/O
- improved security for the execution of the user job script using Slurm
- improved isolation for the nodes using advanced network features of Omni Path (OPA)

In the following, related works will be discussed in Section 2, a general overview of the architecture of HPC systems is provided in Section 3, the design of the secure workflow is presented in Section 4 followed by the description of the actual implementation on our HPC system in Section 5. A security analysis is done in Section 7 followed by benchmarks in Section 8 to measure the overhead of the applied methods. The user story, demonstrating the presented Secure HPC environment in a real use case, i.e. sleep stage scoring, is provided in Section 9 which also includes a comparison of run-times to an insecure workflow. The conclusion is provided in Section 10.

## 2. Related Work

The general need for secure compute capabilities and the resulting requirements for refinement of existing security concepts, particularly for HPC systems, is acknowledged in the literature. Christopher et al. describe in (Christopher, Jung and Doane, 2019) that "At UC Berkeley, this has become a pressing issue" and it has "affected the campus' ability to recruit a new faculty member".

In BioMedIT (Coman Schmid, Crameri, Oesterle, Rinn, Sengstag and Stockinger, 2020), a distributed network is described, where virtualization is used to create completely isolated compute environments that are exclusively reserved for a particular project, in a private cloud. However, the use of virtualization for isolation purposes is only effective, if it is ensured that other users can not access the host system directly. Therefore, this approach requires dedicated hardware and software, which drastically increases the cost of hosting such a system.

A similar approach is described in (Scheerman, Zarrabi, Kruiten, Mogé, Voort, Langedijk, Schoonhoven and Emery, 2021), where *Private Cloud on a Compute Cluster (PCOCC)*[1] is used to deploy a private virtual cluster. The outlined Slurm integration allows for direct integration into an existing HPC system. However, a dedicated Lustre file system is needed and it remains unclear how this virtualized cluster is secured against a compromised host.

Systems like *SELinux* (Smalley, Vance and Salamon, 2001) or *AppArmor* (Bauer, 2006) are using the *Linux Security Modules* to enforce mandatory access control with the general goal to prevent zero-day-exploits. However, once an attacker found a vulnerability these mechanisms become useless. Here, intrusion detection systems, like *auditd*, could be used to detect such an system (Karns, Protin and Wolf, 2012). In contrast, the goal of this paper is to provide a Secure HPC partition under the assumption that a root exploit was already successfully done. Therefore, these two approaches work completely independently of each other.

In order to limit the actions a malicious root can take on a compromised system, one can use kerberized filesystems (Miller, Neuman, Schiller and Saltzer, 1988). This would then, however, limit the functionality of non-interactive batch jobs, and/or would require the additional usage of e.g. *Yubikeys* by all its users.

One possible way to isolate a single task on a multi-tenant node is the use of *Trusted Execution Environments* (TEEs). Here, access to sensitive data or code, which is loaded into memory, is secured from access from the host kernel. There exist several different solutions, including commercial solutions like Intel's SGX (McKeen, Alexandrovich, Anati, Caspi, Johnson, Leslie-Hurd and Rozas, 2016) and open-source solutions like *Keystone* (Lee, Kohlbrenner, Shinde, Asanović and Song, 2020) which are based on basic primitives provided by the respective hardware. In order to utilize those so-called *enclaves*, changes to the source code of the corresponding application are necessary. To mitigate this issue, solutions like *Graphene* (Tsai, Porter and Vij, 2017) enable users to run unchanged code within an enclave. Similarly, *SCONE* (Arnautov, Trach, Gregor, Knauth, Martin, Priebe, Lind, Muthukumaran, O'keeffe, Stillwell et al., 2016) was developed to support Linux secure containers for Docker. These solutions for TEEs are very interesting to secure and isolate a running process, including its data, from malicious access but are in itself not sufficient to provide an end-to-end workflow to securely upload, store and process sensitive data on an untrusted, shared system.

*Containers* are processes which are executed on the host operating system and are pseudo-isolated by *namespaces* and *cgroups*. This allows the provisioning of a private root file system in order to execute software in a portable environment. As with any other process, *containers* are executed with the rights of the user, which can be extended with an *setuid*. A common *container* technology on HPC systems is *Singularity* (Kurtzer, Sochat and Bauer, 2017).

Secure storing of sensitive data on a shared, untrusted storage on an HPC system was explored in (Smith, Riley, Syed, Kupcevic, Edmon and Yockel, 2019). Here, *Ceph Object Gateways* (Weil, Brandt, Miller, Long and Maltzahn, 2006) are deployed on single-tenant compute nodes alongside an *S3FS* which bind-mounts the corresponding S3-Bucket as an POSIX compatible directory onto the host. The host-based configuration then performs automatic encryption/decryption of data that is written/read to/from this specific directory. While this is an important part of secure data processing, it is only a part of a holistic solution that we present. For instance, features such as secure transfer of the necessary keys for accessing the S3-Bucket and for performing the decryption/encryption are important. Additionally,

[1] https://github.com/cea-hpc/pcocc

some data center policies may require a strict separation between the HPC and the storage networks.

Other ways to securely store sensitive data on a typical filesystem include *Linux Unified Key Setup*[2] (LUKS), *eCryptFS*[3], and *GoCryptFS*[4]. LUKS, which is based on `dm_crypt`, provides encryption for a block device and allows for multiple passphrases to be used. On the other hand, eCryptFS and GoCryptFS offer filesystem-level encryption which is based on a FUSE filesystem. To allow for efficient file access while still using a block-cipher, they are splitting each file up into independent blocks, or *extends* as it is called in the case of eCryptFS. The metadata about these blocks are managed by initialization vectors that precede a group of blocks. The advantage is, that random access to a file is faster, since only each involved block needs to be de- or encrypted, instead of the entire file.

In contrast to the related work, we present a blueprint for a holistic end-to-end processing pipeline suited for sensitive data to be used on a shared, untrusted HPC system. This is complemented by a detailed discussion about the security implications as well as extensive benchmarking to assess the general applicability.

## 3. General Usage of HPC systems

This section describes the architecture of HPC systems, as well as the general usage of HPC systems. Following this introduction, a security risk analyzes is performed.

### 3.1. Architecture of HPC systems

Generally, HPC systems are composed of different node types. They serve different purposes and have, therefore, different security policies applied to them. In the following, an overview of typical node types is provided and their interactions are explained. This will further serve as the basis for the nodes which are deemed as secure, even in the case of a privilege escalation of a user.

The general architecture of an HPC system is illustrated in Figure 1. HPC systems are commonly guarded by a perimeter firewall, requiring users to connect via a Virtual Private Network (VPN) or a jump host. Afterwards they can login via *Secure Shell (ssh)* on a *frontend* node. *Frontend* nodes are shared by all users and are used to build software, move data, or submit compute jobs to the batch system. Access to computing resources is granted by a resource manager, like *Slurm* (Yoo, Jette and Grondona, 2003), which schedules user jobs in such a way, that the general utilization of the system is maximized. The *batch system* dispatches jobs to the compute nodes. Although an interactive compute job is generally possible, the majority of the available compute time is consumed by non-interactive jobs, i.e. they run completely without any user interaction. The frontend as well as the compute nodes share at least one parallel file system, like *Lustre* (Braam, 2019) or *BeeGFS* (Herold, Breuner and Heichler, 2014).

[2]https://gitlab.com/cryptsetup/cryptsetup/
[3]https://www.ecryptfs.org/
[4]https://github.com/rfjakob/gocryptfs

The management nodes are comprised of several different nodes that are solely reserved for the admins. Hence, they share the basic requirement, that they need to be protected from any user access. Typically, there is a specific admin node, which is used just for login. Very important for our secure workflow is the so-called image server, i.e., the node which is used to provision the golden images to all the nodes, including the frontend and the compute nodes. If an attacker would gain access to this server, the images could be compromised and distributed to the nodes. In order to increase the security of this node, it is placed in the Level 2 security zone, where access is highly limited and requires further activation and authorization. In order to decrease the attack surface from the image server in the Level 2 security zone, compared to the admin nodes which are being operated in the Level 1 zones, the image server does not allow access from any other system located in a lower security level, i.e. the Level 1 admin nodes and the user nodes. In addition, there are no user accessible daemons listening on any ports. Write access to the images is strictly limited to this image server. The read access for the compute nodes via the Preboot Execution Environment (PXE), necessary for the actual deployment of the images to the nodes, is only done by an admin node located in the Level 1 layer. In this way, a possible exploit in the cluster manager, i. e. the system responsible for distribution the golden images, cannot lead to a compromised image.

Another important node is the one, that the resource manager is running on. This server is responsible to enforce the correct assignment and access of the jobs and users to the compute nodes. The last pieces of infrastructure are the networks that connect the different nodes.

The provisioning of software is usually done via an environment module system, like *Lmod* (McLay, Schulz, Barth and Minyard, 2011) or *Spack* (Gamblin, LeGendre, Collette, Lee, Moody, De Supinski and Futral, 2015), and is cluster wide available, which allows replicating the desired working environment by loading the appropriate modules on any node of the cluster.

### 3.2. Typical User Workflow

The typical workflow to execute a job on an HPC system is depicted in Figure 2. A user logs in and can write or submit a batch script. This is typically similar to a *shell* script, where the desired resources and the commands to be executed are specified. The resource manager checks, whether or not the specified resources are eligible for the *uid* the request is coming from, i.e. if the user is authorized to use the specified resources. If the request is permissible, the resource manager schedules the job in an appropriate time slot for execution with the overarching goal to maximize overall system utilization. The input and output data on a parallel file system can be accessed from all nodes. The necessary communication between the storage nodes and the compute nodes or, in the case of multi-node jobs which are communicating via the Message Passing Interface (MPI), takes place via a high performance interconnects like

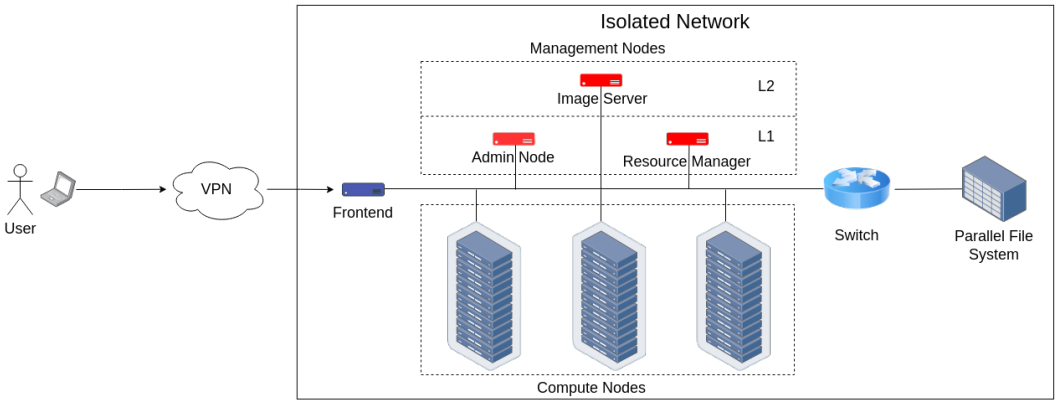


**Figure 1:** A schematic sketch of an HPC system. As shown, HPC systems are typically protected by a perimeter firewall and can only be accessed via a VPN or a jump host. The system consists of frontend, compute, and management nodes, as well as a network and a parallel file system. User access should be restricted to the frontend and the compute nodes, which are shown in blue. The management nodes, shown in red, must be inaccessible for users and from nodes with user access.

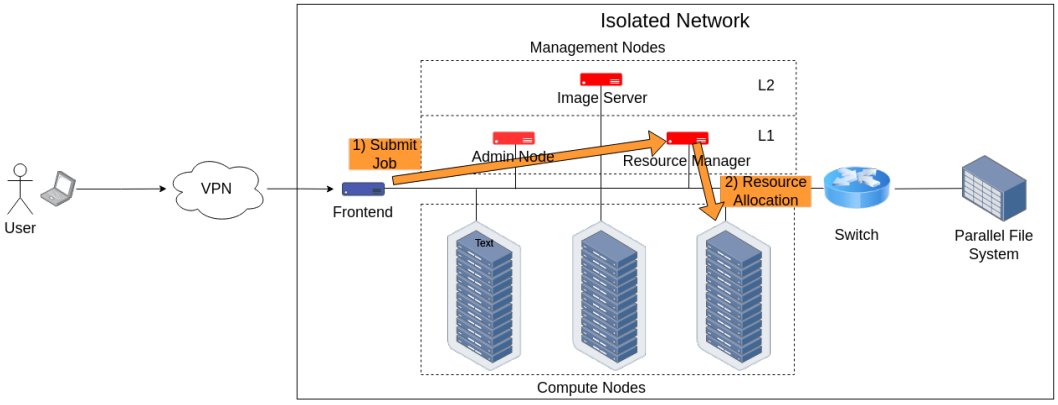


**Figure 2:** A schematic sketch of the typical workflow for users on an HPC system. Usually, users submit a batch script to the resource manager. The access permission of a user to a certain node is hereby solely based on the *uid*. Since data is stored unencrypted on a shared file system and the integrity of the software stack does not have to be guaranteed, the job can start without any further overhead. This workflow is only secured by the user isolation of the utilized nodes Linux kernel.

*Omni-Path* (Birrittella, Debbage, Huggahalli, Kunz, Lovett, Rimmer, Underwood and Zak, 2015) or *Infiniband* (Pfister, 2001).

## 3.3. Possible Attack Scenarios

From the HPC system architecture and user workflow, we can infer potential security risks that we discuss in the following. We assume that a privilege escalation, i.e. a user gaining *root* privileges, can happen on any system accessible to users, in particular the frontend and the compute nodes because the nodes and storage systems users have direct access to are solely protected by the permission system of the *Linux* kernel, the trusted code base is very large and has a large attack surface which presumably yields unknown vulnerabilities (Chen, Mao, Wang, Zhou, Zeldovich and Kaashoek, 2011) that has been exploited in the past.

For the following security analysis, it is therefore assumed, that an attacker can gain *root* access or can impersonate the user on one of these nodes.

### *3.3.1. Data Stored on a Shared File System*

Starting from the node the attacker gained *root* privileges, *root* can get access to any file stored on one of these nodes or is located on a shared file system mounted on this node. This direct access can be made a little bit more uncomfortable, for instance, by an Network File System (NFS) *root-squash* prohibiting direct root access. Such issues can be circumvented by an attacker by changing the *uid*. Therefore, we consider all data to be compromised.

### *3.3.2. Data Stored on a Compute Node*

Additionally, after the job has started, the user is also able to log in on these nodes, for instance via *ssh*. The access is hereby granted by the resource manager solely based on the *uid*. Thus, a *root* user has immediate access to all nodes allocated to users and therefore access to the local data and processes. Furthermore, a *root* user can submit jobs to the batch system with an arbitrary *uid*, thus gaining access to compute nodes reserved by the resource manager for a specific user group.

#### 3.3.3. Manipulation of the Provided Software

A malicious *root* user can also tamper with the provided software stack or with the individual system image of the node the attacker is currently on. Such a compromised system can then continuously leak data.

#### 3.3.4. Manipulation of the high-speed interconnect

When the processing power for application needs to be scaled-up, the application's data is typically divided between multiple processor cores and run as parallel tasks (often called *ranks*), and an MPI library is used to manage the communication of data between the ranks. For small numbers of ranks, this can be on a single host computer; for larger numbers, multiple hosts (often called *nodes*) are used, and the capabilities of the interconnect become important. To get the best scalability, that is the efficiency with which additional cores deliver additional performance, high-speed interconnects like *Omni-Path (OPA)* or *Infiniband* are used in HPC systems to provide better latency, message rate and bandwidth than a conventional Ethernet would.

The MPI data is injected into the high-speed interconnect unencrypted. Encrypting the MPI data, even with hardware, would add latency. Even a minor increase in latency can have a significant impact on scalability, and so encrypting MPI data is undesirable.

In practice, the unencrypted MPI data consists of numerical data exchanged between the ranks of the job as it performs the calculations required by the application. As such, a bad actor managing to access this data is unlikely to be able to derive any value from it. Every node in the job knows the identities of the other nodes, so for a bad actor to partake in the MPI data exchanges, it would need to spoof the identity of a valid node. Therefore, the risk of data leakage, even in an unsecured network, is fairly low.

However, it is always possible that if an unsecured node hosting a bad actor can send data to a secured node, then it may be able to exploit some previously unknown weakness within the secured systems.

These high-speed interconnects are switched networks which rely on a *subnet manager*, also known as a *Fabric Manager (FM)*, for configuration, including the creation of the used routing tables. An attacker can try to imitate the FM on hijacked nodes and bombard the switches with malicious configurations. In addition, an attacker could try to spoof its source node and ingest packages in order to maliciously manipulate the execution of the job on the secure node.

Because the MPI data is unencrypted, it is desirable to apply additional measures to ensure the overall security of the system, and this is discussed in later sections.

## 4. General Design of the Secure Workflow

As motivated before, all systems to which users have direct access could be considered to be compromised and insecure as (unknowingly malicious) software running by a user may have gained administrator permissions. Also, an administrator (Unix root user) should not be considered completely trustworthy and permissions should be limited as much as possible. In order to design the secure workflow, data and software need to be protected on such an exposed system and a mechanism is necessary to trust selected nodes, on which the actual computations can be securely done. Based on the discussed security problems identified in Section 3.3 a secure workflow was designed which mitigates these problems. This secure workflow is presented in the following.

### 4.1. Assumption

In order to provide trust in an otherwise untrusted system, this trust needs to be derived from a secure source system. Therefore, it will be assumed that i) the *image server* of the HPC system as well as ii) the local system of the user, for instance, the respective workstation or laptop, is secure. These assumptions are reasonable because on the one hand the *image server*, as shown in Figure 1, is located within the 2nd security level of a cybersecurity onion of the already highly guarded admin nodes, which deploy only limited services and orchestrate the cluster. On the other hand, the local system of the user is the system where the data resides unencrypted at the beginning of the workflow. Therefore, it has to be secure since otherwise the data would be leaked without any involvement of the secure workflow.

### 4.2. Overview

As discussed in Section 3.3 there are different attack scenarios that a secure workflow has to protect against. These scenarios can be divided into the protection of the data in transit, at rest, and during compute. In order to secure data during transit and at rest, encryption is typically used. To secure data during compute, one needs an ideally air-gaped, but at least an isolated system or node. Based on these two simple ideas a generic secure workflow was developed as depicted in Figure 3.

Unlike in the case of the typical workflow, presented in Figure 2, it is not possible to upload data, containers, and batch scripts directly into the shared file system, since this would leave the workflow vulnerable to attacks depicted in Section 3.3, except if it is encrypted using state-of-the-art encryption. therefore, the first step is to encrypt the data as well as the container on the local system of the user. Afterward, in Step 2 the encrypted data is then uploaded onto the shared storage of the HPC system. The key is uploaded onto a key management system. This communication channel is completely independent of the HPC system and can therefore be considered out-of-band. Analogously should be proceeded with the containers.

In order to be able to retrieve the keys from the key management system, a valid access token needs to be provided in the batch script. To prevent this token from being leaked to an attacker on the frontend, the batch script needs to be encrypted as well. This can happen completely independent of the used resource manager by implementing this mechanism on the secure node, the job should run on. For this, a public-private key pair is created on the system image of the secure node. The public key is then distributed to the users and can be used to encrypt the batch script

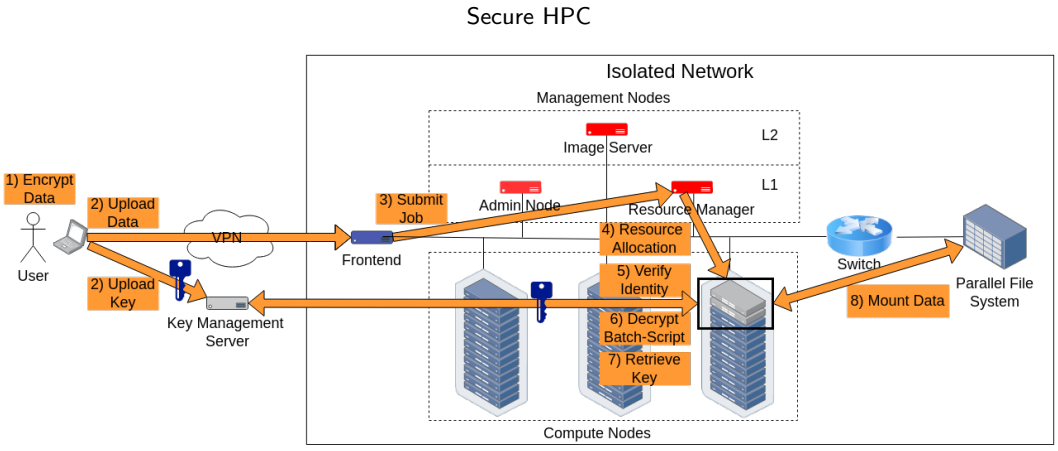


**Figure 3:** A schematic sketch of the secure workflow on an HPC system, which is divided into distinct steps. First, the sensitive data along with the container and the batch script are encrypted on the local machine of the user. Additionally, the batch script is also signed by the user. After encrypting, the components are safe to be uploaded into the shared file system of the HPC system (Step 2). Although the batch script is signed and encrypted, it can be normally submitted in Step 3. As usual, the resource manager will allocate the required resources and start the job in Step 4. In Step 5, the authenticity of the user request is verified and in Step 6, the batch script is decrypted. Using the provided token for the key management system (KMS) the keys are retrieved in Step 7 and used to decrypt the data and container on the isolated, secure node in Step 8.

while the private key has to be highly guarded. Since it is only available on the secure node and on the image server, this mechanism depends on the safety of the latter. This encrypted batch script can be submitted to the resource manager just like any other unencrypted script. For this, a corresponding *decrypt_and_execute* command needs to be implemented on the secure node, which takes as input the encrypted shell script.

The resource allocation made by the resource manager in Step 4 is only dependent on the used *uid* of the user on the HPC system. As discussed in Section 3.3 this is not secure enough, therefore the authenticity of the batch script needs to be checked. In this proposed reference workflow, this problem is solved by the user signing the batch script after it was encrypted with a private key. Here, the corresponding public key has to be made available to the secure node before a job can be submitted.

During Steps 5 and 6 the counterpart of the previously described Step 1 is done. This means, that first the signature of the batch script is checked to verify that this batch script was legitimately submitted. Here, the stored public key of the user the request was made from is used to match the signature. If that verification was successful, the batch script is decrypted, yielding a shell script that can be executed.

Within this script, an access token for the key management is provided. This token is used in Step 7 to retrieve the keys needed to decrypt the data and container. Since by design every legitimate job has to successfully retrieve keys in order to be executed, the success is monitored and the job is killed upon failure.

In Step 8 the keys are used to decrypt the data and container from the shared file system on the secure node. Now, the intended job of the user can be executed within the container. Using a container at this stage probably is the easiest way to maintain a heterogeneous software stack that is required to support the diverse processing steps. As mentioned, the additional advantage here is, that the container image itself can be encrypted and thus the integrity can be ensured, since tampering is only possible if the key is known. Furthermore, mounting all unsafe file systems per default read-only into the container prevents an accidental data leak, for instance by files that are temporarily written by the program without the knowledge of the user.

## 4.3. Securing an OPA Fabric

Broadly, security is applied in two areas: (1) Partitioning the job's nodes from the other nodes in the cluster, and (2) Securing the *Fabric Manager (FM)* from malicious interference.

**Partitioning**: This prevents packets from an unsecured node from reaching a secured node (and packets from a secure node from leaking to an unsecured node).

Omni-Path (in common with Infiniband) uses a system of Partition Keys (PKeys). PKeys are a mechanism used to support Soft Partitions, which enable the creation of multiple, overlapping communication domains. Using PKeys, we can run the secured nodes in a separate partition to the other nodes of the HPC cluster. Within the configuration of the Fabric Manager (FM), we can specify the partitions we require, and the nodes that we want to be members of those partitions.

The FM sends the PKeys over the fabric to the switches and the adapters in the nodes. The FM updates the PKeys whenever there is a change in the FM's PKey configuration. If the switch receives a packet for a node that does not have the required PKey, it will block it. This is a very secure system, implemented in the hardware of the switches and adapters, and can only be controlled by the FM. The adapter

is logically split into a fabric side and a PCIe side; the PKeys are managed from the fabric side, so there is no way that malicious code on the node can change the PKey configuration of the adapter. Therefore, by using PKeys, we can be sure that no unsecured node can send packets to a secured node.

Because a node can be a member of more than one partition (if it has been given more than one PKey), a user needs to specify which PKey to use when running a service that accesses the high-speed interconnect.

For example, to launch an MPI job on PKey 0x0012, the user sets the PSM2_PKEY environment variable to 0x0012.

For the IP network interfaces typically used by storage systems, a network interface device is configured for each PKey that needs to be accessed. From within Linux, these devices behave in the same way as Ethernet devices and each will be configured on a different IP subnet. The network interface running with the default PKey appears as the familiar ib0, and an additional interface on PKey 0x0022 would appear as the device ib0.8022[5].

**Securing the FM**. The FM provides the PKeys to the switches and adapters in the fabric, it is necessary to prevent it from becoming compromised or that some malicious interference with the Fabric Manager can take place.

It is possible that a bad actor on the bare metal of an unsecured node could bring up a competing FM. It would be difficult to do this with predictable results, but the activity could be disruptive. The FM's traffic runs in the Admin vFabric with its own PKey, and in a secure configuration, only the FM node(s), identified by the hardware GUID of its adapter card, are full members of this vFabric. Thus, any FM-like activity from a node that has not been authorized will be blocked.

Therefore, it is important that the FM node (and its standby) are secured, meaning that only the most privileged admin users are able to login to these nodes.

### 4.4. Isolating Compute Nodes

In order to be able to provide secure compute capacity, an isolated node or subcluster is exclusively available to an individual or group for an arbitrary amount of time. These nodes have restrictive firewall settings, the running administrative services are slimmed down, for instance, the usual monitoring is deactivated, and no login capability, e.g. via *ssh*, is available. Furthermore, the nodes need to reliably boot a trustworthy operating system and must enforce that only authorized users can access the node. This is done in two steps: i) Users can only submit jobs that are running in a non-interactive mode, therefore they are required to submit a batch script to the resource manager, and ii) this batch script needs to be signed with a secret on the local system of the user. This secret is a private key, which must never become compromised. The matching public key is available on the secure node to verify the authenticity of the submitted job. Since this secret will never be uploaded to the HPC system, an attacker can't get access to it and can't submit jobs to a secure node from a false *uid*.

Software that is usually provided via environment modules needs to be installed in the system image of the secure node, since using these shared modules would mean importing an untrusted codebase into the secure node.

### 4.5. Key Management System

The usage of the key management system should also follow certain best practices. Although the depicted security measures should be sufficient, it is favorable to use a one-time token mechanism to retrieve the keys from the key management system. If an attacker got their hands on a token, with which the keys can be retrieved, the legitimate request from the user will fail, because the token was already used. Thus, it is immediately obvious that a security incident took place. The token should be short-lived as well, to limit the availability of the keys in the case that a job crashes before the token could be used.

Additionally, we place a reverse proxy in front of the key management system. Here, the received *API* calls can be filtered based on the source *IP* address. The goal is to allow an upload of keys from anywhere, however, limit the legitimate requests to retrieve keys via HTTP get to the secure nodes.

## 5. Implementation of the Secure Workflow

The design and implementation were done with a minimal number of assumptions so that it can be considered as a blueprint for other systems with different requirements as well. Based on the general design presented in Section 4, an implementation was done on an HPC system at GWDG.

### 5.1. Key Management System

*Vault*[6] is used for the key management system (KMS). It allows for the distribution of personal tokens to individual users. With those, users can generate tokens with limited permissions and a short, configurable lifetime. A *response wrapping* is used on these tokens in order to enable single-use tokens to access the deposited keys. In addition, the root token can be reliably deactivated, preventing the root user from spying on the user keys.
In front of *Vault NGINX*[7] with the *ngx_http_geo_module* is deployed as a reverse proxy. It performs IP-address filtering based on the http-verb in order to allow an upload of a key from external systems but restricts the response for the key retrieval to be only sent to a secure node.

### 5.2. Data and Software Management

Since most HPC applications expect a POSIX-IO compatible file system, *Linux Unified Key Setup* (LUKS) was used to encrypt the data. These LUKS containers can be mounted, if the decryption key is available, thus providing the expected interface while transparently encrypting everything written to that mount.

[5] The 0x8000 bit is always set in the device name.

[6] https://www.vaultproject.io/

[7] https://www.nginx.com

| vFabric | Full Members | Limited Members |
|---|---|---|
| General | General | |
| Secure | Secure | |
| Storage | Storage | General, Secure |

**Table 1**
Partitions and their Members

In order to use encrypted containers, *Singularity* is used. Similar to the native LUKS data containers, these encrypted *Singularity* images are decrypted in kernel space as well. This means they reside decrypted in the RAM of the host, thus swapping needs to be deactivated on these secure nodes, to prevent that sensitive data is written unencrypted onto a non-volatile storage medium like a local SSD. By bind mounting only the LUKS data containers into the *Singularity* container, it is ensured that only encrypted write access is possible from the container onto the file system.

## 5.3. Isolating a Secure OPA vFabric

The Omni-Path FM uses a concept of VirtualFabrics (vFabrics) that can be used to create partitioned groups of nodes and/or apply QoS to different classes of traffic. In this discussion, we will focus on partitioning. When used for partitioning, vFabrics use PKeys as the underlying mechanism.

Nodes are assigned membership of partitions within the configuration of the FM. Nodes can be Full Members or Limited Members. Full Members can communicate with all members, but Limited Members can only communicate with Full Members. This feature is useful when nodes in different partitions need access to the same shared resource.

*This description is simplified to remove some of the site-specifics configuration at GWDG.*

- We create 3 DeviceGroups in the FM configuration: General, Secure and Storage. These DeviceGroups hold the identities of the relevant nodes.
- We create 3 vFabric partitions: General, Secure and Storage which contain the corresponding DeviceGroups as Full Members.
- Additionally, we define the General and Secure DeviceGroups to be Limited Members of the Storage vFabric.

In this way, both General and Secure nodes can access the Storage nodes, but General and Secure cannot access each other. The Storage nodes must be secured in a similar way as the FM node(s), as a bad actor on these nodes could get access to the nodes in the Secure vFabric.

*Note:* Currently, the OPA FM is not able to implement the Full/Limited functionality in the GWDG environment, and so a workaround is required using only Full Membership. The workaround provides the same protections but with less flexibility, and will be removed when the functionality is available.

## 5.4. Isolating a Secure Node

In order to isolate a secure node, the system image is hardened. To prevent an attacker from login into that node, a restrictive firewall configuration is used. In addition, suitable services for accessing these nodes, like *ssh*, are turned off. For all services which need to be listening on a specific port, like the *slurmd*, only the IP address of the known counterpart, like the node where the *slurmctld* is running on, is reachable. In order to ensure these settings, a node needs to directly boot into these restrictive configurations and the image server, as well as the network which is used for the PXE boot, need to be trusted. Therefore it is mandatory, that an attacker can not reach the management nodes, and particularly not the level 2 layer of the employed security onion, which was outlined within the made assumptions.

In order to allow for secure inter node communication via our *Omni-Path* Fabric, a secure *vFabric* (virtual Fabric) has to be configured. It is important to disallow the ingestion of management packages from any *HFI* port that is not connected to the dedicated fabric managers. These fabric managers also have to be located within the security onion of the admin nodes. Additionally, the fabric manager needs to be configured to quarantine nodes if they try to spoof their identities, for instance in order to reach into a secure *vFabric*.

## 5.5. Submitting a Batch Job

In order to use encryption for the batch script, a 4096-bit *RSA* (Rivest, Shamir and Adleman, 1978) key pair is created in the system image, and the public key is shared with the user. Since the scheduler might, like Slurm does, require a valid bash-script to be submitted, a small workaround needs to be used. The actual command, one wants to execute is encrypted in a `gpg`[8] message and passed to a helper function located on the secure node, as shown in, yielding a valid bash-script and an effectively encrypted bash-script. Such a batch script is shown with a shortened PGP message in the following listing:

```
#!/bin/bash
/usr/bin/decrypt_and_execute <<EOF
-----BEGIN PGP MESSAGE-----
hQIMA8ErHWKpRkmoAQ/+LyPQJORoQwC5UjqNqXPcebqVYXfqgdt1rPo
[...]
=cRKs
-----END PGP MESSAGE-----
EOF
```

As previously discussed, the authenticity of the compute request needs to be ensured on the secure node. In order to do that, a detached signature of the batch script is created on the local system of the user and is uploaded into the same directory as the batch script (on the HPC system). When submitting the batch script to Slurm via the `sbatch` command, the batch script is parsed and sent to the `slurmctld`. From there it is copied to a secure node using a communication channel between the `slurmd` on the secure node and the `slurmctld`. After this is done, the `slurmd` starts executing the

[8] https://gnupg.org/

job. However, before the just received and locally stored bash-script is invoked, the `Slurmd Prolog` is executed. Since this is the first piece of code that gets triggered to be executed on the secure node, here the detached signature the user-provided is compared to the local copy of the batch script. Only when this detached signature matches, the user-provided code starts to be executed. Upon failure, one can choose to either cancel the job or quarantine the node.

After the batch script is decrypted, the resulting bash-script is executed by the `decrypt_and_execute` method. Here, the provided token is used to get the keys from *Vault*. These keys are then briefly stored in a *tmpfs* to mount the LUKS data containers and to execute the *Singularity* container. Since any legitimate job will require at least two keys, one for the *Singularity* container and one for the LUKS data container, the successful retrieval of the keys is also monitored and mandatory to continue the execution. The keys used to mount the data on the secured node can be deleted afterward. The key for the Singularity container, however, needs to remain for the duration of the execution on the node. Since only the LUKS data containers have a writable bind mount within the *Singularity* containers, results can only be stored there, thus enforcing compliance with data security regulations per design. After the job has finished or was killed by the resource manager *Slurm*, all mounted LUKS data containers are unmounted and the stored key for the Singularity container is deleted from the *tmpfs*. This behavior can be enforced within the *Slurm Epilog*. In the end, the user can download the LUKS output container, where the results are stored for further inspection.

## 6. Extension for a Parallel Secure Workflow

The described secure workflow was so far only intended for single node-jobs. When extending this approach to multi-node jobs, which run for instance via *MPI* (Clarke, Glendinning and Hempel, 1994), one is confronted with three distinct challenges. The first is to ensure secure communication between the nodes running the job. This was already achieved by the previously presented secure OPA configuration. The second challenge is the key distribution. Since there might be multi-node jobs where the number of nodes is not known before the job actually starts, or the orchestration across nodes is difficult and therefore error-prone, it is advantageous to keep the general idea of single-use tokens and distribute them the keys after the successful retrieval. The third challenge is to provide suitable, encrypted parallel I/O.

These aspects are discussed in the following.

### 6.1. Parallel Starter

In order to port our single-node approach to support multi-node applications, we need to distribute the decryption keys to all involved nodes in a secure manner. Since multi-node support requires a secure network to allow for safe MPI communication, one can reuse this network to safely distribute the keys to all nodes. For this, a *Parallel Starter* can be used, which is an MPI-capable program, where only

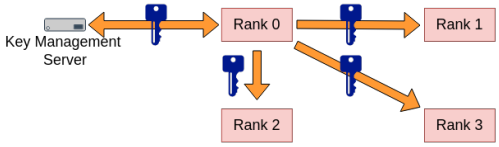


**Figure 4:** Schematic sketch of the parallel starter. Only the process with Rank 0 uses the token from the KMS to retrieve the keys. Afterwards the process with Rank 0 distributes the keys to the other processes using the secured network.

the process with Rank 0 will access the single-use tokens to receive the keys from the key management system. Afterward, the process with Rank 0 uses a broadcast to distribute the keys to all other processes, where then each process needs to acquire a lock, like a node-local semaphore, for the particular node they are located on. This process is depicted in Figure 4. If multiple processes are running on a single node, the first process that gets the lock will write the file to the `/keys` path, which is a `tmpfs`, and initiates the mounting of the encrypted data on the node. The other processes on a shared node will skip this step. This ensures that on a single node the keys are only written once to the `/keys` path and no race condition occurs. Since this is a standalone tool, it is completely independent of the used scheduler. Alternatively, when using Slurm, one can also simply use a `srun -procs-per-node=1`.

It is noteworthy, that there might be other, more specific methods like the `sbcast` tool from Slurm. Although this can be a full-fledged drop-in replacement, one needs to ensure that the communication only takes place inside the secured network. This is guaranteed if the node is properly isolated as described in Section 5.4.

### 6.2. Creating and Managing a Secure Partition

A *Secure Partition* specifies an isolated sub-cluster within the used HPC system. This sub-cluster is also managed by the resource manager, e.g. Slurm, within a private partition where only predefined users can submit to. In addition, the nodes which comprise the secure partition can communicate with each other over a high-speed interconnnect, e.g. OPA. This is securely possible since the secure nodes of a secure partition are all within a dedicated vFabric. This means, that no traffic is being routed from an unsecure node to a secure node, therefor enforcing network isolation of the secure partition on the switch level. Of course, all of the nodes in a secure partition are booted into the same secure system image. In summary, all changes necessary to create a secure partition can be done in software, rendering an additional procurement unnecessary.

Since only software and configuration changes are necessary, switching nodes from unsecure to secure operation can be done dynamically to match the provided resources to the actual demand. For this, a script can be executed on the admin node which automatically changes the configuration of the resource manager, the cluster manager, the fabric

manager, the reverse proxy settings on the key management system to adjust the ip filtering of the http verb, and triggers a reboot of the corresponding machine to boot it into the secure/unsecure system image. Due to this high degree of automation, the partitioning can be done quickly without much effort by the administrators. The required time to reconfigure a secure partition as described above is mainly determined by the time a node needs to reboot.

### 6.3. Parallel Input/Output

The concurrent write access to a LUKS container from multiple nodes is not possible by design and would corrupt the file system. Read-only mounts are supported as they prevent the modification of the file system, therefore, the input containers could be mounted on all nodes.

We assume the amount of data to be written by the application is unknown prior execution but can be bounded (e.g. 10 GiB). Hence, a LUKS container with the maximum expected size can be created as a file with holes and formatted[9]. Thus, the actual occupied capacity of the LUKS container is small regardless of its size. For example, a 10 GiB container file formatted with ext4 requires 69 MiB of space and 85 MiB if created with LUKS and then formatted with ext4.

In the following, when we mention usability this refers to the overall complexity introduced. We could provide tools that automate and simplify the handling for any of these cases, still some difficulty would remain for the application.

In order to support parallel write, there are various design alternatives with individual advantages:

*Single rank I/O with an encrypted LUKS container (SL).* Only one rank in particular Rank 0 writes to a dedicated output container. *Pros:* the container could be automatically created and mounted; no change of applications with traditional (non-parallel I/O) is necessary; the output container can be fetched by the user and directly be mounted. *Cons:* It does not allow output from multiple nodes, hence performance scalability is limited. This strategy is suitable for small volumes of output.

*Parallel I/O to independent LUKS containers (PIL).* Here, the ranks of each node write output to the node-local LUKS containers. *Pros:* utilizes a parallel file systems with the maximum performance and provides a node-local metadata cache. *Cons:* usability, the number of containers is dependent on the number of nodes - making it difficult to gather and aggregate the results for the user. This may require adjustment of the application and post-processing toolchain. This strategy is useful for cases with demanding I/O and a robust post-processing pipeline.

*Parallel I/O to encrypted files with keys stored in a LUKS container (PFKL)* Each process writes the data directly to the parallel file system but they encrypt each file (individually). Practically, at program runtime, the processes decide upon an encryption key and Rank 0 stores the key for this file in the initially prepared LUKS container where it can later be fetched by the user. A file might be created collectively (requires coordination of the key) or individually. *Pros:* individual keys for each file increase security. Compatible with independent or shared files. *Cons:* difficult to handle key retrieval in the post-processing and data analysis. Either the I/O path or the application must support encryption. This strategy is not advised due to the introduced complexity.

*Parallel I/O to encrypted files with pre-shared key (PFSK)* In this scenario, the user would embed a key into the application or retrieve it from *Vault* that then is used for the encryption of any created files. *Pros:* individual keys for each file increase security. Compatible with independent or shared files. *Cons:* the I/O path or the application must support encryption. As the key is known by the user a-priori, this strategy is superior to PFKL while integrating the advantages of the other methods.

*Parallel I/O with an overlay file system with pre-shared key (POSK)* In this scenario, a user would start a MPI job, where Rank 0 would fetch the key from Vault. This key is then distributed across all nodes and an encrypted mount is created on all compute nodes using a stacked cryptographic file system like eCryptFS, GoCryptFS, EncFS[10], or fscrypt[11]. *Pros:* easy way to provide a POSIX compliant file system. No support of application needed. Minimal barrier for the user. *Cons:* user must be cautious to not write files unencrypted, i.e. to a wrong path[12]. Parallel I/O to a shared file is not possible.

A qualitative comparison of the different methods is given in Table 2. Compatibility refers to the support I/O modes in the application and complexity to the overall setup and resulting artifacts. PFSK is currently our favorite solution, however, to achieve optimal compatibility the transparent encryption in the I/O path or the utilized I/O library is necessary. POSK is a good alternative but we need to spend more time on it to prevent users to accidentally store data on an unencrypted directory. Also modifications to the stacked file system to support parallel I/O to a shared file must be made.

Files with holes are a problem with ecryptfs, as a file that is seeked beyond current EOF and written is completely filled by ecryptfs.

## 7. Security Analysis

Based on the general design, presented in sec. 4, and the actual implementation, presented in sec. 5, a concluding assessment of possible attack scenarios along with their

[9] For example using the command `$ dd if=/dev/zero of=/tmp/test bs=1024 count=1 seek=$((1024*1024*10))`.

[10] https://github.com/vgough/encfs

[11] https://github.com/google/fscrypt

[12] This could be prevented (in the future) by only making the encrypted mounts accessible.

| Method | Performance | Compatibility | Usability | Complexity |
|---|---|---|---|---|
| SL | - | - | + | + |
| PIL | ++ | - | - | - |
| PFKL | + | + | - | - |
| PFSK | ++ | + | + | 0 |
| POSK | + | + | ++ | + |

**Table 2**
Comparison of different storage strategies

respective mitigation strategies presented before, is done in this section.

### 7.1. Man-in-the-Middle attack

A man-in-the-middle attack can happen in this secure workflow during the execution of Step 2, as shown in Figure 3. One can see, that on the one side, a man-in-the-middle attack can happen during the communication with *Vault*. This communication is done via the provided *Rest API* and is secured via TLS.On the other side, an interception of packages can also happen during the upload of data to the HPC system. Here, data is secure since it was encrypted on the client-side and the communication itself is guarded via *ssh*.

In both cases, the attacker would end up with state-of-the-art encrypted data, which can't be used without the corresponding decryption key. As presented, these are highly guarded and only retrievable for authorized users. Thus, access to the network infrastructure outside of the HPC system can't diminish the security of this workflow.

### 7.2. Privilege Escalation

A user only uploads encrypted data and encrypted *Singularity* containers, thus the attacker can neither gain access to the decrypted data nor can the software environment that accesses the data directly be compromised. The same argument holds for the submitted batch scripts. These are encrypted as well and thus ensure the confidentiality of the token of the key management system.

As discussed, a *root* user can submit jobs from the *uid* of a legitimate user. This can neither be prevented by the kernel nor by the resource manager relying on the kernel. The obvious mitigation would be a multi-factor authentication which is prompted upon the submission of a batch script by a trusted management server. This, however, needs to be supported by the individual resource management software in use. A resource manager independent way was presented before, where the batch script needs to be signed by an *S/MIME* certificate.

To summarize, a *root* user can neither get access to the decrypted data, tamper with the software or system image, and can not impersonate a user on the system.

### 7.3. IP-Spoofing

In order to prevent that an attacker can retrieve the keys stored in *Vault* with a stolen token, *Nginx* was used as a reverse proxy in front of *Vault*, in order to filter out GET requests from an IP address, which is not a secure node. This is configured on the key management system and to change that, access to this system is required, including access to the administrative network where the *ssh* port is available. An attacker can, however, use a false source IP address and mimic that the request was done from a secure node. Then, *Vault* would send the requested keys but would do so to the specified secure node. Thus an attacker would still need to get access to such a hardened node.

### 7.4. User Operating Errors

Since the presented secure workflow has quite some steps which a user has to execute correctly to ensure the integrity of the processing, mistakes can happen and potentially impair the security measures. In order to simplify the application for a user, wrapper scripts are provided, which, for instance, automatically create and mount LUKS containers on the local system of a user while using strong random passwords. Furthermore, it is ensured, that the created keys are only uploaded to our *Vault* instance, and not accidentally on an untrusted system. Lastly, once a user has written locally a batch script that is ready for submission, a script can be used locally, to encrypt, sign, upload, and submit the batch script.

### 7.5. Network Manipulations

Depending on the used high-speed interconnect, which is typically used in HPC systems, like *Omni-Path* or *Infiniband*, there are additional threats associated. In Section 5.4 it was discussed that a *Omni-Path* fabric can be securely locked down to ensure reliable operation even in the case of a privilege escalation on the connected, user-accessible nodes.

## 8. Performance Analysis

In order to determine the performance costs when switching from the unsecured workflow depicted in Section 3.2 to the secure workflow presented in Section 4 and Section 5, different benchmarks have been done. These benchmarks can be roughly divided into two distinct groups. One type of benchmark is designed to quantify the static overhead associated with the secure workflow, while the other measures the dynamic cost of the used encryption. The performance measurements of the encryption is done once for a single node with LUKS and once with multiple nodes on an eCryptFS and GoCryptFS stacked filesystem. Since the secure nodes are otherwise isolated, there are no additional costs during compute.

### 8.1. Measuring Encryption Costs

Encryption and decryption take place during write and read operations to a storage device, like a parallel file system. In order to simulate different I/O patterns to get a better understanding of the potential performance decrease, the *IO500* (Kunkel, Bent, Lofstead and Markomanolis, 2016) benchmark was used. The encryption was done with *AES512* in the case of LUKS (Moh'd, Jararweh and Tawalbeh, 2011) and with AES-256 in the case of eCryptFS and GoCryptFS.

| Operation (unit) | Performance | |
|---|---|---|
| | Encrypted | Unencrypted |
| *ior-easy-write* [GiB/s] | 0.6 | 2.8 |
| *mdtest-easy-write* [kIOPS] | 15.2 | 24.4 |
| *ior-hard-write* [GiB/s] | 0.06 | 0.01 |
| *mdtest-hard-write* [kIOPS] | 15.9 | 6.2 |
| *find* [kIOPS] | 270.8 | 211.8 |
| *ior-easy-read* [GiB/s] | 0.6 | 2.2 |
| *mdtest-easy-stat* [kIOPS] | 194.0 | 121.1 |
| *ior-hard-read* [GiB/s] | 0.3 | 0.4 |
| *mdtest-hard-stat* [kIOPS] | 69.6 | 44.4 |
| *mdtest-easy-delete* [kIOPS] | 22.6 | 33.4 |
| *mdtest-hard-read* [kIOPS] | 0.7 | 2.1 |
| *mdtest-hard-delete* [kIOPS] | 17.8 | 3.5 |

**Table 3**
*IO500* results on *BeeGFS*

#### *8.1.1. Performance Comparison on the Parallel File System on a single node with LUKS*

As discussed in Section 4, the typical use case for the secure workflow is assumed to be that users upload their encrypted data onto the shared parallel file system and only decrypt them on the secure nodes. In order to measure the performance costs, two different scenarios are benchmarked. In the first case, an unsecured workflow is used, where an unencrypted *Singularity* container executes the before mentioned *IO500* benchmark on a native bind mount on the parallel file system. In the second case an encrypted LUKS container, using *cryptsetup*, is mounted locally on the node with a loopback device. The latter case represents the secure workflow, therefore also an encrypted *Singularity* container is used to perform the *IO500* benchmark. Both container images in these two benchmarks were created using the same *Recipe*-file.

The benchmarks were done on a dedicated node of the *Scientific Compute Cluster* hosted by *GWDG*. It features an Intel Xeon Platinum 9242 CPU with 376G of DDR4 memory operating at 2934 MT/s and runs on an *3.10.0-1160.36.2.el7.x86_64 Linux* Kernel. The used filesystem runs *BeeGFS* and has 4 metadata servers and 14 storage servers. The node is connected to the *BeegFS* storage via a 100 Gb/s *OPA* fabric.

Before the benchmarks has been started, 343G of the 376G available memory has been filled up and the swap was deactivated. The *LUKS* container was opened via *cryptsetup 2.3.3* and was mounted as an *ext4* file system.

The results of the performed benchmarks are presented in Table 3. The first observation here is, that the *ior-easy-write*, which is sensible to streaming performance, reaches in the encrypted case only $\approx$ 23% of the bandwidth of the unencrypted case. For the other operations it is much harder to interpret these results correctly. The reason is, that two concurrent effects which are close to impossible to disentangle have a influence on the performance. The first effect comes from the previously mentioned encryption which is done by `dm_crypt`, which therefor leads to a decreased read/write performance when being flushed out of the page cache. The second effect is, that the unencrypted IO500 run uses for its metadata operations the metadata servers of the BeeGFS cluster. The encrypted IO500 run, however, does (almost all) metadata operations on the local node, since it has locally mounted an `ext4` filesystem. For the BeeGFS filesystem this LUKS container containing this `ext4` filesystem is only one large file. The metadata operations, which are primarily benchmarked in the different `mdtest` runs, are therefor handled by the local node itself, not by the metadata servers from the BeeGFS filesystem. This problem becomes even greater when compairing the `ior-hard` runs, since here 47008 Bytes are written/read/stat/deleted. During the `mdtest-easy` runs no data is written, and during the `mdtest-hard` runs only 3901 Bytes are written. Therefor, it is tricky to directly compare the metadata performance between those two runs.

In summary, one can observe a non-negligible performance degradation, particularly during streaming IO, when compared to the unencrypted measurement. This can be seen in the operations containing an *easy*.

#### *8.1.2. Analysis of Cryptsetup*

A recent analysis of the *dm-crypt* implementation found that the different work queues used to enable asynchronous processing of I/O requests can actually drastically slow done performance. To circumvent this problem, *dm-crypt* can be instructed to avoid a queuing of IO requests and execute them synchronously. This feature was merged into the *Linux* Kernel in version 5.9[13].

In order to further analyze the origin of the previously observed performance difference between the encrypted and the unencrypted use case, the kernel of the used node was updated to the most recent version 5.16.3, and *cryptsetup 2.4.3* was compiled from source. Since the clients for the parallel file systems of the *Scientific Compute Cluster* do not support newer kernel versions, the performance difference could only be measured on the node. For this, a *tmpfs* was used, which has the additional advantage of offering the lowest latency and highest bandwidth. This means, that any additional overhead can not be hidden by bottlenecks located on the storage device. The file for the *loopback device* had a size of 340G of the available 376G. In order to support the *vader BTL* of *OpenMPI* (Gabriel, Fagg, Bosilca, Angskun, Dongarra, Squyres, Sahay, Kambadur, Barrett, Lumsdaine et al., 2004) additional 10G was provided in a *tmpfs*.

The results of these measurements can be seen in Table 4. Since in both cases, i. e. the encrypted and the unencrypted ones, a `ext4` filesystem residing in RAM is mounted via a loop back device. This means, that unlike in the case before, here also the performance of the metadata operations can be compared, since all deviations can be accounted to the overhead of `dm_crypt`. One can see, that there are only slight differences, however no clear trend can be recognized.

One important observation is that it can be confirmed that using encrypted *Singularity* containers does not have

[13] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=39d42fa96ba1b7d2544db3f8ed5da8fb0d5cb877

| Operation [unit] | Synchronous | | | | Asynchronous | | | |
|---|---|---|---|---|---|---|---|---|
| | 10 Processes | | 1 Process | | 10 Processes | | 1 Process | |
| | Encrypted | Unencrypted | Encrypted | Unencrypted | Encrypted | Unencrypted | Encrypted | Unencrypted |
| *ior-easy-write* [GiB/s] | 1.2 | 1.2 | 1.1 | 1.0 | 1.6 | 1.7 | 1.0 | 1.0 |
| *mdtest-easy-write* [kIOPS] | 111.7 | 123.3 | 70.0 | 70.6 | 111.4 | 111.5 | 69.0 | 69.0 |
| *ior-hard-write* [GiB/s] | 0.6 | 0.6 | 1.1 | 1.1 | 0.7 | 0.7 | 1.1 | 1.0 |
| *mdtest-hard-write* [kIOPS] | 18.8 | 19.8 | 31.6 | 33.9 | 15.5 | 15.3 | 31.1 | 35.6 |
| *find* [kIOPS] | 3821.4 | 3858.6 | 1493.4 | 1460.7 | 7093.1 | 5847.2 | 1456.0 | 1490.6 |
| *ior-easy-read* [GiB/s] | 1.0 | 1.1 | 1.0 | 1.0 | 2.1 | 1.8 | 1.3 | 1.3 |
| *mdtest-easy-stat* [kIOPS] | 558.9 | 537.9 | 183.5 | 180.0 | 566.7 | 567.1 | 179.1 | 181.6 |
| *ior-hard-read* [GiB/s] | 1.3 | 1.3 | 1.4 | 1.5 | 1.9 | 1.9 | 1.5 | 1.5 |
| *mdtest-hard-stat* [kIOPS] | 391.7 | 422.5 | 186.4 | 186.7 | 448.3 | 403.7 | 184.8 | 187.6 |
| *mdtest-easy-delete* [kIOPS] | 78.3 | 74.2 | 102.6 | 103.3 | 81.4 | 82.2 | 103.5 | 102.4 |
| *mdtest-hard-read* [kIOPS] | 188.0 | 180.7 | 48.5 | 49.2 | 213.8 | 209.3 | 45.7 | 46.2 |
| *mdtest-hard-delete* [kIOPS] | 64.9 | 62.4 | 75.3 | 79.4 | 63.2 | 60.7 | 78.6 | 73.7 |

**Table 4**
Results of the IO500 benchmark on an encrypted *LUKS* container residing in a *tmpfs*. The specification Encrypted and Unencrypted refers to the Singularity container

| Operation (unit) | Performance | |
|---|---|---|
| | 1 Process | 10 Processes |
| *ior-easy-write* [GiB/s] | 0.9 | 2.1 |
| *mdtest-easy-write* [kIOPS] | 68.9 | 131.0 |
| *ior-hard-write* [GiB/s] | 0.8 | 0.8 |
| *mdtest-hard-write* [kIOPS] | 32.4 | 30.8 |
| *find* [kIOPS] | 1391.2 | 5832.1 |
| *ior-easy-read* [GiB/s] | 2.1 | 2.6 |
| *mdtest-easy-stat* [kIOPS] | 180.3 | 584.3 |
| *ior-hard-read* [GiB/s] | 2.6 | 2.0 |
| *mdtest-hard-stat* [kIOPS] | 173.8 | 380.4 |
| *mdtest-easy-delete* [kIOPS] | 98.7 | 72.4 |
| *mdtest-hard-read* [kIOPS] | 72.8 | 206.9 |
| *mdtest-hard-delete* [kIOPS] | 77.5 | 61.4 |

**Table 5**
*IO500* results on an *ext4* mounted loopback device residing in an *tmpfs*

any measurable performance impact at runtime. The second observation is, that in this particular case one profits from an asynchronous execution of the encrypted IO operations during parallelized execution with 10 processes. This can clearly be seen in the highlighted cells containing the results from the streaming intensive *ior-easy-write* and *ior-easy-read* operations as well as in the *ior-hard-read* test. The reason for this could be the use of *loopback devices* and *device mappers*, which causes differences in the execution of the IO requests at the block device level, when compared to a natively encrypted block device, like a hard drive or an SSD.

In order to estimate the actual encryption cost, a baseline for an unencrypted scenario was measured, wherein the exact similar setup of the same file was mounted as an *ext4* file system with an *loopback device* without the usage of *cryptsetup*. The results are shown in Table 5. By comparing the performance increase when scaling from 1 process to 10, there is still very limited scalability exhibited. The source for this issue is assumed to lie within the usage of a *loopback device*. Comparing the results of the *ior-easy-write*, where by far the most data is being written and therefore is mostly hit by the cryptographic overhead one can see that by comparing to the asynchronous test in Table 4 $\approx$ 80% performance was achieved. The achieved value of $\approx$ 2.1 GiB/s is very close to the value of $\approx$ 2.3 GiB/s one obtains when running the provided benchmark suite of *cryptsetup*.

In summary, one can clearly see a performance advantage of newer *Linux* kernels, however, it was not possible to replicate the advantage of synchronous cryptographic IO execution on this system.

### *8.1.3. Performance Comparison on the Parallel File System on multiple nodes with eCryptFS*

In order to extend the secure workflow to multi-node support, and to offer alternatives to LUKS, also filesystem-level encryption is examined. For this eCryptFS was set up on 10 nodes, which were otherwise identical to the system described in Section 8.1.1. On these 10 nodes, the same directory on the shared parallel filesystems, i.e. the previously described BeeGFS cluster, was mounted using the same passphrase. The specified symmetric cipher is AES with an 32-byte key and activated filename encryption. Unlike the OpenPGP standard, eCryptfs allows for random access in a single file (Halcrow, 2005). To this end, eCryptFS breaks a file into different distinct parts so called *extents*, which have been encrypted with a cipher operating in block chaining mode. Therefore, any read or write operation only requires the entire extent to be decrypted, not the entire file. This however also entails, that parallel IO of multiple processes from multiple nodes needs to respect this offset which is defined by the size of the extends. Unlike the case of an unencrypted file system, it is not enough to define a non-overlapping offset among all processes to ensure undisturbed file access, but this non-overlapping needs to be ensured on an extend level.

Therefore, the IO500 configuration needs to be adapted, to accommodate this issue. Here, all shared file I/O was disabled to ensure an undisturbed run.

The results of this reduced run are shown in Table 6. One can see nearly linear scalability in streaming operations, i.e. primarily ior-easy-write and ior-easy-read, when scaling

| Operation (unit) | eCryptFS | | | GoCryptFS | | | | Native | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Nodes, Procs. per Node | 1,1 | 1,10 | 10,1 | 1,1 | 1,10 | 10,1 | 80,10 | 1,1 | 1,10 | 10,1 | 80,10 |
| *ior-easy-write* [GiB/s] | 0.4 | 1.5 | 3.4 | 0.2 | 1.6 | 1.6 | 6.0 | 1.4 | 3.8 | 3.8 | 9.2 |
| *mdtest-easy-write* [kIOPS] | 1.9 | 7.6 | 13.4 | 1.9 | 11.8 | 15.5 | 59.4 | 5 | 25.9 | 31.3 | 83.5 |
| *mdtest-hard-write* [kIOPS] | 1.7 | 2.7 | 6.7 | 1.05 | 2 | 5.7 | 7.8 | 2.4 | 5.1 | 7.5 | 12.0 |
| *find* [kIOPS] | 83.1 | 271.8 | 28.3 | 12.9 | 34.5 | 53.2 | 156.1 | 74.9 | 251.2 | 268.4 | 966.1 |
| *ior-easy-read* [GiB/s] | 0.4 | 1.5 | 2.5 | 0.4 | 1.4 | 2.5 | 3.2 | 0.5 | 2 | 2.9 | 8.7 |
| *mdtest-easy-stat* [kIOPS] | 20.5 | 98.3 | 4.6 | 10.6 | 30 | 84.1 | 374.1 | 15.4 | 124.4 | 137.8 | 392.7 |
| *mdtest-hard-stat* [kIOPS] | 20.6 | 56.8 | 2 | 9.1 | 9.3 | 76.4 | 75.1 | 17.9 | 55.5 | 112.8 | 138.6 |
| *mdtest-easy-delete* [kIOPS] | 5.1 | 8.8 | 1.9 | 3 | 17.3 | 16.4 | 53.1 | 7.1 | 31.1 | 32.3 | 57.5 |
| *mdtest-hard-read* [kIOPS] | 0.2 | 2 | 1.2 | 0.3 | 3.2 | 2.5 | 11.8 | 0.4 | 2.1 | 2.4 | 23.8 |
| *mdtest-hard-delete* [kIOPS] | 4.4 | 3.6 | 1.2 | 2.3 | 2.8 | 6 | 10.5 | 3.9 | 2.8 | 4.6 | 13.3 |

**Table 6**
*IO500* results of an eCryptFS layer on top of a BeeGFS cluster compared to a native BeeGFS mount.

from 1 node to 10. Please note, that in Table 6 the ior-easy-write for one process on one node was rounded up from 0.35 GiB/s.

When comparing the performance of the eCryptFS stacked filesystem against the native BeeGFS mount, one can see a performance drop of 75% when comparing the performance of the ior-easy-write for a single process on a single node. However, when scaling out to 10 processes on a single node, this difference was already reduced to a 60% performance drop. When using 1 process per node and 10 nodes the performance difference is reduced further to 8% performance penalty. These values are obtained by dividing the unrounded values from the eCryptFS rows with the matching ones from the Native table. The rather constant performance difference in the `mdtest-easy-write` comes from the mismatch of the 3901 Byte write operation compared to the size of an extent, since always the entire needs to be processed for the block-cipher. Since there is a slightly higher performance drop when there are 10 processes on a single node, there might also be the effect of shared, and stacked caches, as described by Wang, Wen, Kong and Yi (2012). Similar arguments hold for the reverse operation, i.e. the read.

#### *8.1.4. Performance Comparison on the Parallel File System on multiple nodes with GoCryptFS*

GoCryptFS also provides file-based encryption based on a FUSE filesystem and was inspired by EncFS. It encrypts files using AES-256. Similar to eCryptFS, it also splits a file up into individual blocks of 4KiB in size and generates a 128-bit Initialization Vector. Additionally, filenames are also encrypted using AES-256. Analog to the previous analysis in Section 8.1.3, the IO500 benchmark was run with 1 process on a node, with 10 processes on a node, with 1 process, and 10 nodes, and with 10 processes on 80 nodes.

Comparing the performance of eCryptFS with GoCryptFS one can see a clear advantage of eCryptFS when comparing the scaling from one node to ten nodes. Here, the single process and the multi-node `ior-easy-write` performance is twice as large, while the saturation on a single node with multiple processes is similar. However, GoCryptFS exhibits still scalability when extending from 10 nodes to 80. Although it can not catch up to the performance of the native BeeGFS mount, it performance reasonable results with two thirds of the native performance, while still having a more predictable behaviour when scaling since it does not depend on the process idstribution on the different nodes as much as eCryptFS does.

When comparing the small file IO performance, e.g. `mdtest-easy-write` with the native BeeGFS client in Table 6 one can see again the performance drop caused by a mixture of the mismatch of the 3901 Byte which gets written compared to the block size of 4096 Bytes of a single block and the general encryption overhead. The first has the effect, that after the first file of 3901 Bytes, always an entire block, in total 4096 Bytes, needs to be first read or written by the filesystem and afterward need to be decrypted and encrypted.

### 8.2. Measuring the Static Overhead

In order to determine the static overhead of this secure workflow, a node was booted into the secure image and the workflow was executed 1000 times. The static overhead contains the verification of the signature and the consecutive decryption of the batch script, the retrieval of the keys from *Vault*, the mounting, umounting of the *LUKS* containers, decrypting and starting the *Singularity* container, and the deletion of the keys residing in memory. The reference job is executing `sleep 10` on bare metal as baseline and within an encrypted singularity container for the secure workflow measurements, and the complete wall clock time was measured with *time*. This job was submitted 1000 times with the normal workflow discussed in Section 3.2 and 1000 times with the secure workflow as implemented in Section 5. For each job, 3 keys had to be retrieved, one for the *Singularity* container, one for the *LUKS* container with the input data, and another one to store the output data in. Both *LUKS* containers have a size of 20 GB. The batch script, which needs to be decrypted, has a size of 544 bytes unencrypted. The result of the benchmark is obtained by subtracting the average amount of the 1000 normal submissions from the individual wallclock time spent in the secure submission. The resulting distribution is shown in Figure 5. One can see that it follows a normal distribution with an expectation

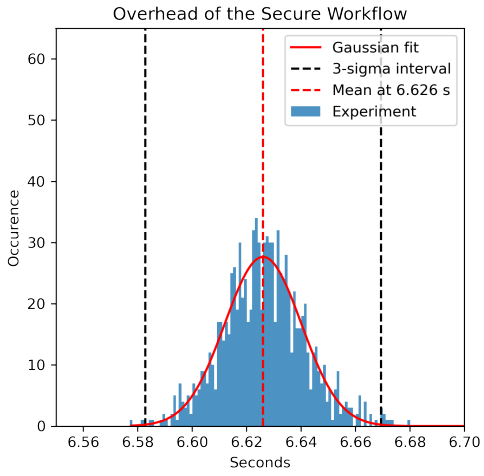


**Figure 5:** Distribution of the individual static overhead measurements of the secure workflow when compared to the same job executed with the normal workflow.

value at 6.63 s and a 3 sigma limit of $\pm$ 0.04 s. This overhead is negligible compared to a typical runtime of 10s of minutes to several hours.

## 9. User Story

This section describes a real use case, where the secure workflow was used. The intent is to further illustrate a concrete setup of an end-to-end secure workflow, with a particular focus on the client machine, and to compare the performance of the secure workflow with a bare metal system on a real use case and hereby offer additional inside into its applicability compared to the synthetic IO500 benchmark. For this a machine learning use case was chosen, since those are typically heavily relying on IO performance. This use case resembles the case discussed in Section 8.1.1, where in the naive setup without any optimizations a performance drop of $\approx$ 75% in streaming performance was observed.

### 9.1. General Description

In order to compare the secure workflow introduced in Section 4 to an unsecured workflow as described in Section 3.2, we evaluate their performance using a typical example from life science. We choose the task of sleep stage assessment by analysis of nocturnal polysomnography (PSG) data. This data is acquired in sleep laboratories and is inspected usually by medical experts who divide the data into epochs of 30 seconds and assign a class to each epoch depending on the depth of sleep (Berry, Brooks, Gamaldo, Harding, Lloyd, Marcus and Vaughn, 2015). Due to the large amounts of data this is a tedious and costly task, and – although the sleep classifications are indented to provide an objective assessment – there is still room for subjective interpretation (Zhang, Dong, Kantelhardt, Li, Zhao, Garcia, Glos, Penzel and Han, 2015). This raised the interest in developing algorithms for faster and more objective classification.

PSG recordings recordings typically include different biosignals such as electroencephalography (EEG), electrooculography (EOG), electromyography (EMG), and electrocardiography (ECG) which are measured continuously. These signals not only contain information about the patient's health status but can also be used to identify him/her (Biel, Pettersson, Philipson and Wide, 2001) (Wang, Hu and Abbass, 2020) which makes the protection of this data highly relevant.

For our experiment, we use data of the SIESTA project (Klosh, Kemp, Penzel, Schlogl, Rappelsberger, Trenker, Gruber, Zeithofer, Saletu, Herrmann, Himanen, Kunz, Barbanoj, Roschke, Varri and Dorffner, 2001) which includes recordings of 98 patients with sleep disorders (e.g. sleep apnea, periodic limb movement syndrome) and 194 healthy controls. The data was recorded in different European sleep laboratories and is stored European Data Format (EDF) with a total file size of 104 Gigabytes. We randomly pick 100 EDF files for analysis.

Sleep stages classification is performed using the pretrained "Stanford Stages" algorithm (Stephansen, Olesen, Olsen, Ambati, Leary, Moore, Carrillo, Lin, Han, Yan, Sun, Dauvilliers, Scholz, Barateau, Hogl, Stefani, Hong, Kim, Pizza, Plazzi, Vandi, Antelmi, Perrin, Kuna, Schweitzer, Kushida, Peppard, Sorensen, Jennum and Mignot, 2018) which consists of three independent convolution neural networks for EEG, EOG, and EMG data which are jointly fed into a long short-term memory network. The final output layer assigns the labels "wake", "Stage 1 sleep", "Stage 2 sleep", "Stage 3/4 sleep", "Rapid eye movement (REM) sleep", or "Unscored" to 15 second intervals. This algorithm was pre-trained on multiple thousands PSGs stemming from 12 different sleep centers in diverse recording environments and following different protocols.

We use the source code of the master branch[14], apply `conda` for installing dependencies as described in the README file, and define a custom JSON file containing our configuration. The algorithm processes each EDF file separately and generates a visual output termed "hypnodensity plot" showing time on the x- and probability of each class on the y-axis. Fig. 6 shows this visualization exemplary for one night (7.5h) of a single patient. The y-axis shows the probability distribution per sleep stage and thereby allows to assess the certainty of the algorithm: Columns with constant colors (stages) depict time ranges where the algorithm has a high certainty and the more colors (stages) there are in a column, the lower the certainty. A higher level of uncertainty can typically be observed in transition between stages and is also present when medical experts annotate data. For the data depicted in Fig. 6, typical patterns of sleep become visible with phases of deep sleep in the first half of the night, more often REM sleep in the second half of the night, and phases of brief awakenings in the early morning.

[14] https://github.com/Stanford-STAGES/stanford-stages

### 9.2. Setup of the Secure Workflow

For illustration purposes and reproducibility, we provide the code of this use case in our GitHub repository [15]. As shown in Figure 3, the starting point of the secure workflow is a safe client machine. In this particular setup a virtual machine (VM) in an OpenStack environment (Sefraoui, Aissaoui, Eleuldj et al., 2012) was used to simulate this client machine. Once the described input data is copied into a LUKS container using a provided helper script, this LUKS container can be uploaded to a specific path in the user's scratch space. An additional empty LUKS container for the output data is created and uploaded. The two keys remain securely on the local VM. A completely analogous procedure is followed with the encrypted Singularity container, which is built once and is then uploaded into the user's home.
The remainder of the secure workflow is processed by a fully automated script. The job-specific commands, i.e. the generalized batch script that should be executed on the HPC systems, are written into a single file called `command.sh.template`. After this job-specific file has been written, the automated secure workflow script can be executed. Here, `command.sh` is created as a copy of the template. Then the keys are uploaded and the necessary single-use tokens are generated in Vault, which are then automatically inserted into the `command.sh` using `sed`. Afterwards the `command.sh` script is encrypted and copied into a `run.sh` as shown in Section 5.5. Lastly, using `gpg -detach-sign` a detached signature is created and stored as `run.sh.sig`. Both files, `run.sh` and `run.sh.sig` are copied to the HPC system into the same folder, which must be readable for `root`. Since Slurm is used in this particular use case, `sbatch` was used to submit the job to a *secure* partition. After the job is finished, the output LUKS container is downloaded to the secure client machine, in this case, the beforementioned VM, using scp and the LUKS container is mounted locally to get access to the data.

It should be mentioned, that the creation of the input data LUKS container was not part of this fully automated process, since the job was run on the same input data multiple times. Generally, the creation and uploading of the input data container can be part of this automated script to offer a complete end-to-end secure processing pipeline.

### 9.3. Results

Fig. 7 shows a comparison of run times when using an unsecured workflow (max: 302s, min: 208s, median:213.5) vs. the secure workflow (max: 338s, min: 212s, median:220). Run time was measured using differences in time stamps of the hyponedensity plots. As can be seen, results are bimodal with two clusters with the majority of run times being in the first within the interval $208 - 250$s. The second cluster begins at around 300s and represents larger EDF files due to longer night sleep. In 79% the secure workflow has a longer run time, in 4% it is equal, and in 17% the secure workflow is faster than the insecure one. This is a clear indication, that the runtime of at least 40% of all runs is so close to each other that statistical fluctuations of the system performance are the dominating factor, not the overhead of the secure workflow. When comparing the medians one can see that the secure workflow takes $\approx 3\%$ longer than the insecure workflow.

Considering that the secure workflow enables the processing of sensitive data and therefore creates the opportunity to reuse a shared and existing HPC cluster for analysis of patient data which – due to privacy constraints – would be impossible otherwise, a 3% performance drop seems fair.

[15] https://github.com/gwdg/secure-hpc

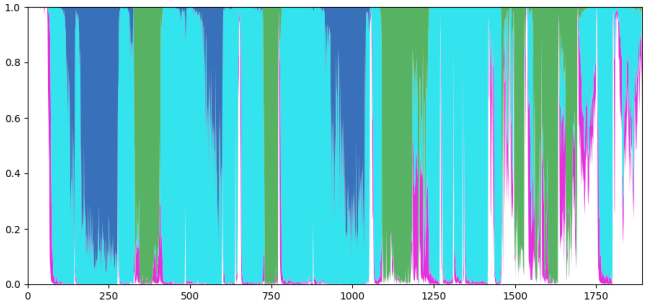


**Figure 6:** Resulting hypnodensity plot for a single night with values on the x-axis representing 15s windows. Stages are "wake" (white), "Stage 1 sleep" (pink), "Stage 2 sleep" (turquoise), "Stage 3/4 sleep" (blue), "REM sleep" (green).

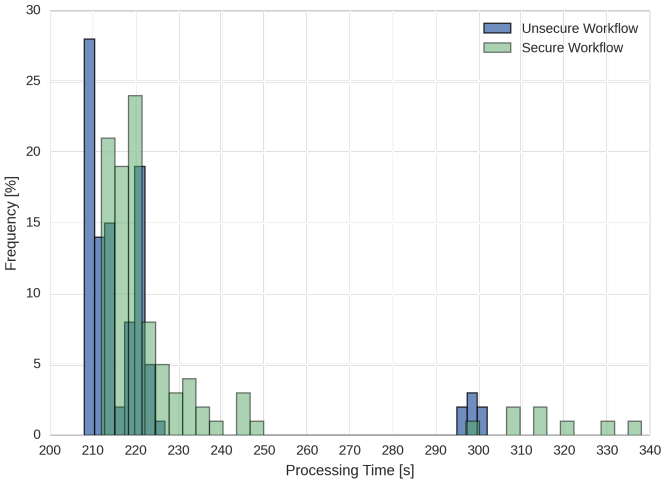


**Figure 7:** Distribution of the running times when processing single nights for sleep stage classification using the secure workflow compared to the insecure workflow.

## 10. Discussion & Future Work

In conclusion, a secure workflow for HPC systems is presented which enables the processing of sensitive data on an existing, untrusted system. This presented workflow can serve as a blueprint for other systems. An in-depth security analysis is discussed based on our actual implementation at GWDG. Extensive benchmarking of three different cryptographic software stacks is done, clearly revealing the cryptographic cost. Then the worst-case scenario is used to showcase an end-to-end automated workflow performing on automatic sleep stage scoring using machine learning.

Here 97% efficiency was reached on average, highlighting the applicability of the presented work.

In future work, we want to look closer at the discussed TEE. The current solutions require a reboot and dedicated public/private keys for each individual group working with sensitive data. Although a privilege escalation is still possible by a member of these groups, this person can only gain access to data already available to him/her. Using TEEs would additionally shield independent processes from each other and could potentially allow to share a node between groups.

On the other side, one needs to closely evaluate the cryptographic costs caused by the need for encryption. As soon as all required file system clients are available for newer kernels, the impact on the *BeeGFS* performance will be tested. Additionally, the performance costs of *loopback devices* should be further analyzed.

Lastly, only a reduced subset of the IO500 benchmark could be run, due to the mismatch of the random access offset and the size of the blocks/extends in eCryptFS/GoCryptFS. Here, some work needs to be done on these stacked file systems to enable parallel IO to shared files and thereby offering the full capabilities of the underlying parallel file systems.

## Acknowledgements

We gratefully acknowledge funding by the "Niedersachsisches Vorab" funding line of the Volkswagen Foundation and "Nationales Hochleistungsrechnen" (NHR), a network of computing centers in Germany to provide computing capacity and promote methodological skills.

## References

Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'keeffe, D., Stillwell, M.L., et al., 2016. {SCONE}: Secure linux containers with intel {SGX}, in: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), pp. 689–703.

Bauer, M., 2006. Paranoid penguin: an introduction to novell apparmor. Linux Journal 2006, 13.

Berry, R.B., Brooks, R., Gamaldo, C.E., Harding, S.M., Lloyd, R.M., Marcus, C.L., Vaughn, B.V., 2015. The AASM Manual for the Scoring of Sleep and Associated Events: Rules, Terminology and Technical Specifications. Illinois: American Academy of Sleep Medicine.

Biel, L., Pettersson, O., Philipson, L., Wide, P., 2001. Ecg analysis: a new approach in human identification. IEEE Transactions on Instrumentation and Measurement 50, 808–812.

Birrittella, M.S., Debbage, M., Huggahalli, R., Kunz, J., Lovett, T., Rimmer, T., Underwood, K.D., Zak, R.C., 2015. Intel® omni-path architecture: Enabling scalable, high performance fabrics, in: 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects, IEEE. pp. 1–9.

Braam, P., 2019. The lustre storage architecture. arXiv preprint arXiv:1903.01955 .

Chen, H., Mao, Y., Wang, X., Zhou, D., Zeldovich, N., Kaashoek, M.F., 2011. Linux kernel vulnerabilities: State-of-the-art defenses and open problems, in: Proceedings of the Second Asia-Pacific Workshop on Systems, pp. 1–5.

Christopher, J., Jung, G., Doane, C., 2019. Making it more secure: The technical and social challenges of expanding the functionality of an existing hpc cluster to meet university and federal data security requirements, in: Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning), pp. 1–5.

Clarke, L., Glendinning, I., Hempel, R., 1994. The mpi message passing interface standard, in: Programming environments for massively parallel distributed systems. Springer, pp. 213–218.

Coman Schmid, D., Crameri, K., Oesterle, S., Rinn, B., Sengstag, T., Stockinger, H., 2020. Sphn–the biomedit network: A secure it platform for research with sensitive human data. Digital Personalized Health and Medicine 270, 1170–1174.

Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., et al., 2004. Open mpi: Goals, concept, and design of a next generation mpi implementation, in: European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting, Springer. pp. 97–104.

Gamblin, T., LeGendre, M., Collette, M.R., Lee, G.L., Moody, A., De Supinski, B.R., Futral, S., 2015. The spack package manager: bringing order to hpc software chaos, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–12.

Halcrow, M.A., 2005. ecryptfs: An enterprise-class encrypted filesystem for linux, in: Proceedings of the 2005 Linux Symposium, pp. 201–218.

Hammernik, K., Klatzer, T., Kobler, E., Recht, M.P., Sodickson, D.K., Pock, T., Knoll, F., 2018. Learning a variational network for reconstruction of accelerated mri data. Magnetic resonance in medicine 79, 3055–3071.

Herold, F., Breuner, S., Heichler, J., 2014. An introduction to beegfs.

Jattke, P., van der Veen, V., Frigo, P., Gunter, S., Razavi, K., . Blacksmith: Scalable rowhammering in the frequency domain .

Karns, D., Protin, K., Wolf, J., 2012. iSSH v. Auditd: Intrusion Detection in High Performance Computing. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

Klosh, G., Kemp, B., Penzel, T., Schlogl, A., Rappelsberger, P., Trenker, E., Gruber, G., Zeithofer, J., Saletu, B., Herrmann, W., Himanen, S., Kunz, D., Barbanoj, M., Roschke, J., Varri, A., Dorffner, G., 2001. The siesta project polygraphic and clinical database. IEEE Engineering in Medicine and Biology Magazine 20, 51–57. doi:10.1109/51.932725.

Kunkel, J., Bent, J., Lofstead, J., Markomanolis, G.S., 2016. Establishing the io-500 benchmark. White Paper .

Kurtzer, G.M., Sochat, V., Bauer, M.W., 2017. Singularity: Scientific containers for mobility of compute. PloS one 12, e0177459.

Lee, D., Kohlbrenner, D., Shinde, S., Asanović, K., Song, D., 2020. Keystone: An open framework for architecting trusted execution environments, in: Proceedings of the Fifteenth European Conference on Computer Systems, pp. 1–16.

McKeen, F., Alexandrovich, I., Anati, I., Caspi, D., Johnson, S., Leslie-Hurd, R., Rozas, C., 2016. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave, in: Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, pp. 1–9.

McLay, R., Schulz, K.W., Barth, W.L., Minyard, T., 2011. Best practices for the deployment and management of production hpc clusters, in: SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE. pp. 1–11.

Miller, S.P., Neuman, B.C., Schiller, J.I., Saltzer, J.H., 1988. Kerberos authentication and authorization system, in: In Project Athena Technical Plan, Citeseer.

Moh'd, A., Jararweh, Y., Tawalbeh, L., 2011. Aes-512: 512-bit advanced encryption standard algorithm design and evaluation, in: 2011 7th International Conference on Information Assurance and Security (IAS), IEEE. pp. 292–297.

Nolte, H., Sarmiento, S., Ehlers, T., Kunkel, J., 2022. A secure workflow for shared hpc systems, in: 22nd International Symposium on Cluster, Cloud and Internet Computing (CCGrid).

Pfister, G.F., 2001. An introduction to the infiniband architecture. High performance mass storage and parallel I/O 42, 10.

Rivest, R.L., Shamir, A., Adleman, L., 1978. A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM 21, 120–126.

Scheerman, M., Zarrabi, N., Kruiten, M., Mogé, M., Voort, L., Langedijk, A., Schoonhoven, R., Emery, T., 2021. Secure platform for processing sensitive data on shared hpc systems. arXiv preprint arXiv:2103.14679 .

Sefraoui, O., Aissaoui, M., Eleuldj, M., et al., 2012. Openstack: toward an open-source solution for cloud computing. International Journal of Computer Applications 55, 38–42.

Smalley, S., Vance, C., Salamon, W., 2001. Implementing selinux as a linux security module. NAI Labs Report 1, 139.

Smith, A., Riley, J., Syed, M., Kupcevic, M., Edmon, P., Yockel, S., 2019. Exploring untrusted distributed storage for high performance computing, in: Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning), pp. 1–6.

Stephansen, J.B., Olesen, A.N., Olsen, M., Ambati, A., Leary, E.B., Moore, H.E., Carrillo, O., Lin, L., Han, F., Yan, H., Sun, Y.L., Dauvilliers, Y., Scholz, S., Barateau, L., Hogl, B., Stefani, A., Hong, S.C., Kim, T.W., Pizza, F., Plazzi, G., Vandi, S., Antelmi, E., Perrin, D., Kuna, S.T., Schweitzer, P.K., Kushida, C., Peppard, P.E., Sorensen, H.B.D., Jennum, P., Mignot, E., 2018. Neural network analysis of sleep stages enables efficient diagnosis of narcolepsy. Nature Communications 9. doi:10.1038/s41467-018-07229-3.

Tsai, C.C., Porter, D.E., Vij, M., 2017. Graphene-sgx: A practical library {OS} for unmodified applications on {SGX}, in: 2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17), pp. 645–658.

Uecker, M., Ong, F., Tamir, J.I., Bahri, D., Virtue, P., Cheng, J.Y., Zhang, T., Lustig, M., 2015. Berkeley advanced reconstruction toolbox, in: Proc. Intl. Soc. Mag. Reson. Med.

Wang, L., Wen, Y., Kong, J., Yi, X., 2012. Optimizing ecryptfs for better performance and security, in: Linux Symposium, Citeseer. p. 137.

Wang, M., Hu, J., Abbass, H.A., 2020. Brainprint: Eeg biometric identification based on analyzing brain connectivity graphs. Pattern Recognition 105, 107381.

Weil, S.A., Brandt, S.A., Miller, E.L., Long, D.D., Maltzahn, C., 2006. Ceph: A scalable, high-performance distributed file system, in: Proceedings of the 7th symposium on Operating systems design and implementation, pp. 307–320.

Yoo, A.B., Jette, M.A., Grondona, M., 2003. Slurm: Simple linux utility for resource management, in: Workshop on job scheduling strategies for parallel processing, Springer. pp. 44–60.

Zhang, X., Dong, X., Kantelhardt, J., Li, J., Zhao, L., Garcia, C., Glos, M., Penzel, T., Han, F., 2015. Process and outcome for international reliability in sleep scoring. Sleep and Breathing 19, 191–5.