A Secure Workflow for Shared HPC Systems

1st Hendrik Nolte
Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen
Göttingen, Germany
hendrik.nolte@gwdg.de

2nd Simon Hernan Sarmiento Sabater
Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen
Göttingen, Germany

3rd Tim Ehlers
Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen
Göttingen, Germany

4th Julian Kunkel
Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen
Göttingen, Germany

***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 a generic and secure workflow that can be implemented on typical HPC systems allowing users to transfer, store and analyze sensitive data. In our experiments, we see an advantage in the asynchronous execution of IO requests, while reaching 80% of the ideal performance.**

***Index Terms*—high performance computing, sensitive data, secure computing**

I. Introduction

The increasing adaption of data and compute-intensive algorithms in digital humanities or life sciences [1], [2] has drastically increased the demand for cost-effective solutions in research domains that are subjected to very strict data security restrictions, like GDPR or 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 [3], 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, even in the case of a privilege escalation leading to a compromised cluster, the integrity of data should be guaranteed.

In this work, we

- present a generic workflow to transfer, store and process sensitive data
- discuss the implementation on the HPC system at GWDG
- provide benchmark results to measure performance

In the following, related works will be discussed in Section II, a general overview of the architecture of HPC systems is provided in Section III, the design of the secure workflow is presented in Section IV followed by the description of the actual implementation on our HPC system in Section V. A security analysis is done in Section VI followed by benchmarks in Section VII to measure the overhead of the applied methods. The conclusion is provided in Section VIII.

II. 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 [4] 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 [5], 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.

We gratefully acknowledge funding by the “Niedersachsis-ches Vorab” funding line of the Volkswagen Foundation and “Nationales Hochleistungsrechnen” (NHR), a network of eight computing centers in Germany to provide computing capacity and promote methodological skills.

A similar approach is described in [6], 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.

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 [7] and open-source solutions like *Keystone* [8] 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* [9] enable users to run unchanged code within an enclave. Similarly, *SCONE* [10] 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 is in itself not sufficient to provide an end-to-end workflow to securely upload, store and process sensitive data on an untrusted, shared system.

Secure storing of sensitive data on a shared, untrusted storage on an HPC system was explored in [11]. Here, *Ceph Object Gateways* [12] 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. However, it remains unclear, how the necessary keys for accessing the S3-Bucket and for performing the decryption/encryption are secured from a privilege escalation on the corresponding nodes. Additionally, some data center policies may require a strict separation between the HPC and the storage networks.

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* [13].

In contrast to the presented related work, we present a blueprint for end-to-end processing of sensitive data on a shared, untrusted HPC system and discuss all security implications in detail.

III. General Usage of HPC systems

A. 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 VPN or a jump host. Afterwards they can login via a *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* [14], 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* [15] or *BeeGFS* [16].

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 only possible from the admin node and requires further activation and authorization, like an additional 2 factor authentication.

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* [17] or *Spack* [18], and is cluster wide available, which allows replicating the desired working environment by loading the appropriate modules on any node of the cluster.

B. Typical User Workflow

The typical workflow to execute a job on an HPC system is depicted in fig. 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

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

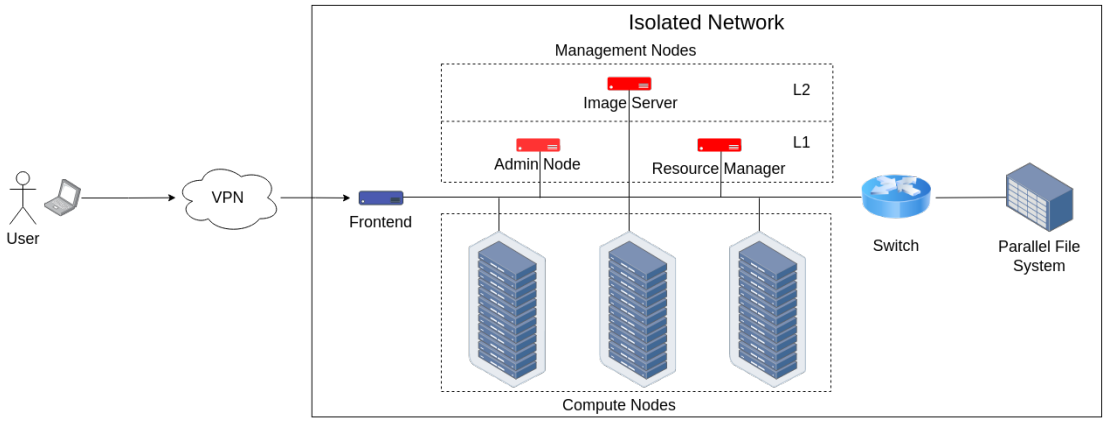

Fig. 1. A schematic sketch of an HPC system. As shown, HPC systems are typically protected by a perimeter firewall and can only be accessed via a VPN or a jump host. The system consists of frontend, compute, and management nodes, as well as a network and a parallel file system. User access should be 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.

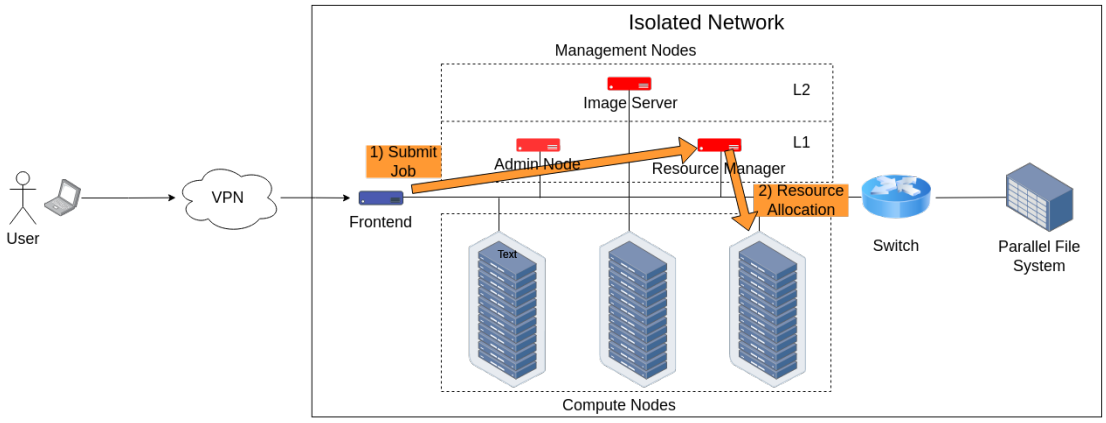

Fig. 2. A schematic sketch of the typical workflow for users on an HPC system. As shown, usually users submit a batch script to the resource manager. The access permission of a user to a certain node is hereby solely based on the *uid*. Since data is stored unencrypted on a shared file system and the 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 host operating system, e.g. the *Linux* kernel.

schedules the job in an appropriate time slot for execution with the overarching goal to maximize overall system utilization.

Needed input and output data on the 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 *MPI*, takes place via a high performance interconnects like *Omni-Path* [19] or *Infiniband* [20].

C. Possible Attack Scenarios

From this general architecture of HPC systems as well as the resulting workflow for users which is geared towards performance, certain security risks are present or are being introduced. Since 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 therefore a large attack surface which presumably yields unknown vulnerabilities [21]. therefore, it is assumed that a privilege escalation, i.e. a user gains *root* privileges, can happen on any system accessible to users, which are the frontend and the compute nodes. For the following security analysis, it is therefore assumed, that an attacker has successfully gained *root* access on one of these nodes.

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 by a *root-squash* for an attacker since now the *uid* needs to be changed, but all data has to be considered compromised.

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) 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.

4) Network Manipulations: In order to provide maximal bandwidth at a minimal latency, high-speed interconnects like *OPA* or *Infiniband*, which are switched networks, are used in HPC systems. These switches rely on a *subnet manager* for configuration, including the creation of the used routing tables. An attacker can try to imitate these *subnet manager* 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.

IV. 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 III-C a secure workflow was designed which mitigates these problems. This secure workflow is presented in the following.

A. 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 fig. 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.

B. Overview

As discussed in section III-C 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 fig. 3.

Unlike in the case of the typical workflow, presented in fig. 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 III-C, 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 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 III-C 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.

Within the last shown step 8) the keys are used to decrypt the data and container from the shared file system on the secure

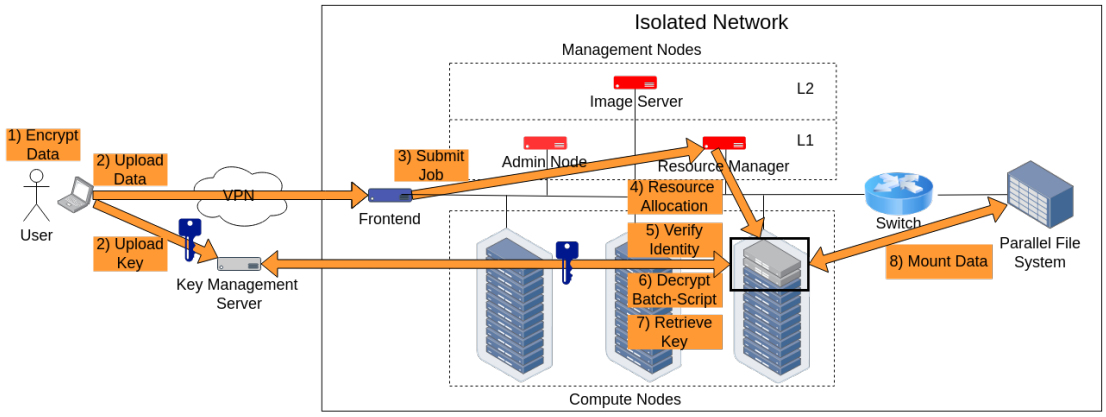

Fig. 3. A schematic sketch of the secure workflow on an HPC system, which is divided into 8 distinct steps. As shown, the sensitive data along with the container and the batch script are encrypted on the local machine of the user. Additionally, the batch script is also signed by the user. After encrypting, the components are safe to be uploaded into the shared file system of the HPC system. Although the batch script is signed and encrypted, it can be normally submitted in the third step. As usual, the resource manager will allocate the required resources and start the job in step 4). In the fifth step, the authenticity of the user request is verified and in the 6th step, 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.

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.

C. 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, as described before. 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.

D. 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, a reverse proxy can be placed 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 get requests only from the secure nodes.

V. Implementation of the Secure Workflow

The design as well as this actual implementation were done with a minimal amount 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 IV, an implementation was done on an HPC system at GWDG.

A. Key Management System

Vault[2] is used for the key management system. 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

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

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*[3] 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.

B. 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.

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.

C. Isolating a Secure Node

In order to isolate a secure node, the system image is adapted. 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*.

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

D. Submitting a Batch Job

In order to use encryption for the batch script, a 4096-bit *RSA* [22] key pair is created in the system image, and the public key is shared with the user. Since also a signature from the user on the batch script is required to prove the authenticity of the submit, an *S/MIME* certificate is used. Using *S/MIME* has the advantage that the existing infrastructure for authentication of the user and the distribution of the certificate can be reused. This workflow is usually in place at compute centers to allow for signed or encrypted E-Mails.

After the batch script is decrypted, the provided token is used to get the keys from *Vault*. These are then only shortly stored in a *tmpfs* to mount the LUKS data containers and to execute the *Singularity* container. Since any legit 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.

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 keys are deleted from the *tmpfs*. This behavior can be enforced within the *Slurm Epilog*. At the end, the user can download the LUKS data container, where the results are stored for further inspection.

VI. Security Analysis

Based on the general design, presented in sec. IV, and the actual implementation, presented in sec. V, a concluding assessment of possible attack scenarios along with their respective mitigation strategies presented before, is done in this section.

A. 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 fig. 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.

B. 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.

C. 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 node, which is highly secured as depicted before.

D. 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.

E. 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 V-C 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.

VII. Performance Analysis

In order to determine the performance costs when switching from the unsecured workflow depicted in section III-B to the secure workflow presented in section IV and section V, 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. Since the secure nodes are otherwise isolated, there are no additional costs during compute.

A. 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* [23] benchmark was used. The encryption was always done with AES512 [24].

1) Performance Comparison on the Parallel File System: As discussed in section IV, 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 I. 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. Similarly, the encrypted *mdtest-easy-write* also only reaches $\approx 60\%$ of the unencrypted performance. The significantly higher performance, which was achieved in the following **-hard* tests can be explained by the fact that due to

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 I
IO500 RESULTS ON *BeeGFS*

the limited overall IO performance a larger percentage could be cached in the remaining 33G of RAM.

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*.

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[4].

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* [25] additional 10G was provided in a *tmpfs*.

The results of these measurements can be seen in table II. One important observation is that it could be confirmed that using encrypted *Singularity* containers does not have 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 III. 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 II $\approx 80\%$ performance was achieved. The achieved value of ≈ 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.

B. 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 a *sleep 10*, was executed bare metal for the reference measurements 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 III-B and 1000 times with the secure workflow as implemented in section V. 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 unencryptedly. 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 fig. 4. One can see that it follows a normal distribution with an expectation value at 6.63 s and a 3 sigma limit of $\pm$ 0.04 s. This overhead is completely negligible compared to a typical runtime of multiple hours for HPC jobs.

VIII. 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

[4]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 II

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 III

IO500 RESULTS ON AN *ext4* MOUNTED LOOPBACK DEVICE RESIDING IN AN *tmpfs*

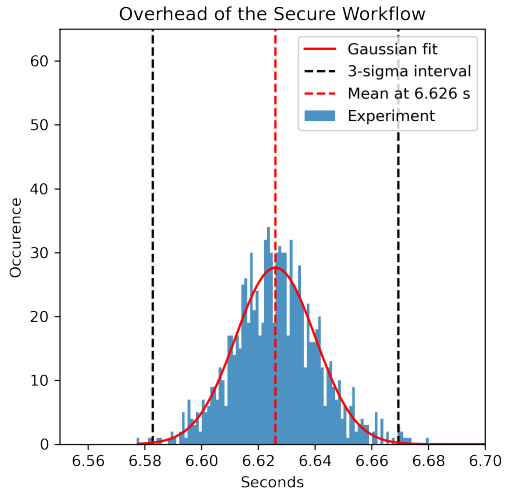

Fig. 4. Distribution of the individual static overhead measurements of the secure workflow when compared to the same job processed insecurely.

analysis is discussed based on our actual implementation at GWDG.

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.

Moreover, this paper has a focus on single-node jobs. In the future, we want to extend this to multi-node jobs and develop automated management tools to manage the secure *vFabric* and images.

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.

REFERENCES

[1] M. Uecker, F. Ong, J. I. Tamir, D. Bahri, P. Virtue, J. Y. Cheng, T. Zhang, and M. Lustig, "Berkeley advanced reconstruction toolbox," in *Proc. Intl. Soc. Mag. Reson. Med*, vol. 23, no. 2486, 2015.

[2] K. Hammernik, T. Klatzer, E. Kobler, M. P. Recht, D. K. Sodickson, T. Pock, and F. Knoll, "Learning a variational network for reconstruction of accelerated mri data," *Magnetic resonance in medicine*, vol. 79, no. 6, pp. 3055–3071, 2018.

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

[4] J. Christopher, G. Jung, and C. Doane, "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)*, 2019, pp. 1–5.

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

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

[7] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "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*, 2016, pp. 1–9.

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

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

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

[11] A. Smith, J. Riley, M. Syed, M. Kupcevic, P. Edmon, and S. Yockel, "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)*, 2019, pp. 1–6.

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

[13] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PloS one*, vol. 12, no. 5, p. e0177459, 2017.

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

[15] P. Braam, "The lustre storage architecture," *arXiv preprint arXiv:1903.01955*, 2019.

[16] F. Herold, S. Breuner, and J. Heichler, "An introduction to beegfs," 2014.

[17] R. McLay, K. W. Schulz, W. L. Barth, and T. Minyard, "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, 2011, pp. 1–11.

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

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

[20] G. F. Pfister, "An introduction to the infiniband architecture," *High performance mass storage and parallel I/O*, vol. 42, no. 617-632, p. 10, 2001.

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

[22] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[23] J. Kunkel, J. Bent, J. Lofstead, and G. S. Markomanolis, "Establishing the io-500 benchmark," *White Paper*, 2016.

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

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