

## Summary of *Securing hard drives with the Security Protocol and Data Model (SPDM)*

The goal of this article is to demonstrate a working prototype of an SDPM-enabled hard drive, evaluating its protected communication with the operating system. To assess the overhead introduced by SPDM (version 1.1.0) in the scenario, they tested their solution against an unmodified, unsecured hard disk on an emulated environment.

### Project design

Securing the communication between the operating system and a peripheral essentially requires altering the device's firmware and writing a matching device driver.

In order to provide a proof-of-concept, they used a virtualized environment with QEMU in a standard Linux box as the host OS, a customized Linux distribution (based on *Buildroot*) as the guest OS and the *virtio* disk.

### Implementations details

#### SPDM integration

To implement the functionalities, they used *libspdm*. Since the kernel has a reduced stack space, the library had to be modified to address this limitation.

The device driver was modified to encrypt/decrypt the write/read requests payload using the symmetric keys established with SPDM. Moreover, a header had to be included to indicate an SPDM encrypted message. Additionally, the size of a single request from the block layer would often exceed the maximum limit allowed by the *libspdm* API. This issue was addressed by repacking the original requests into smaller requests of adequate size and encoding pure SPDM messages with a special request type.

#### Corner cases

At both endpoints, *libspdm* internally maintains a monotonically increasing 64-bit counter that is used as a nonce. Hence, message decryption would fail whenever the order of messages sent/received differed from the encryption order. To address this issue, they modified the counter's behavior by relying on the fact that the 16 least significant bits of this 64-bit counter are placed at the message header, using it directly while keeping track of 16-bit overflows (to ensure the nonce's non-repeatable nature).

In addition, the request repackaging strategy at the device driver level has increased the load on an internal data pool used to allocate new requests. So, when the pool is depleted and the device driver attempts to allocate a new request, the underlying request allocation function attempts to execute the pending requests. If the next pending request is the one that triggered the new allocation (due to the repackaging strategy), a loop is created, eventually leading to a stack overflow. To address this issue, they have increased the capacity of the default data pool.

## Experiment results

The experiments involved measuring the time taken to copy files of different sizes to/from a partition on a virtio disk equipped with SPDM. For reference, the same procedures were performed on an unmodified virtio disk.

The files were filled with random data, for sizes varying from 5KiB, 10KiB, 50KiB, up to 100MiB, and each configuration was executed 30 times, each time after clearing the system cache.

The results are:

- The impact of SPDM security is negligible for small file sizes (up to 5k). This implies other overheads (e.g., physically moving disk heads) are predominant in this setting.
- The gap between the secured and unsecured system begins to be noticeable for medium-sized files.
- The performance of the secured hard disk degrades as file size increases, reaching up to 7 times slower transference procedure.
- Virtio to virtio transactions show the largest performance drop, since the data must be encrypted twice in that case: from disk to OS, and then from OS to disk.

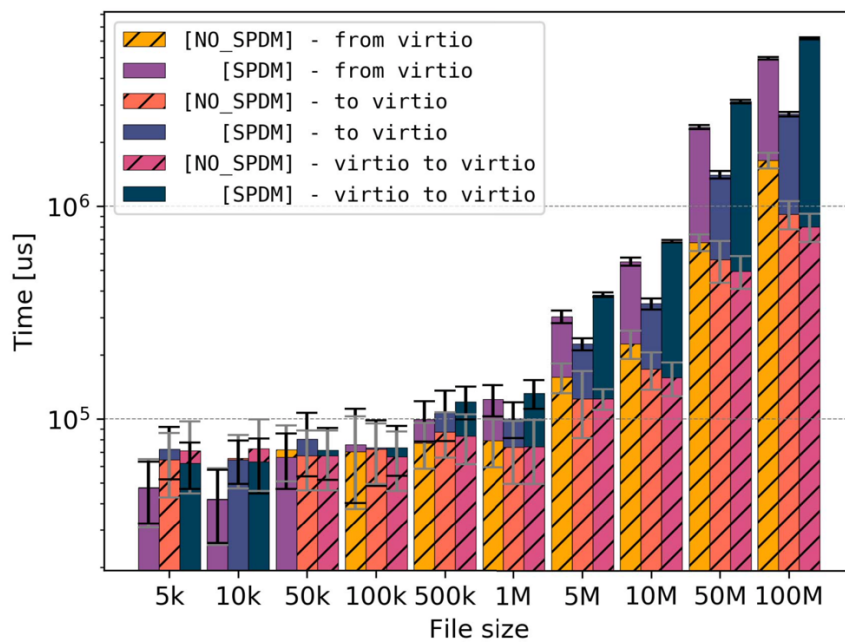


Fig. 2. Time to copy files of varying sizes in different settings