International Journal of Information Security (2025) 24:126
https://doi.org/10.1007/s10207-025-01034-y

REGULAR CONTRIBUTION

Benchmarking the security protocol and data model (SPDM) for component authentication

Renan C. A. Alves[1] · Bruno C. Albertini[1] · Marcos A. Simplicio Jr.[1]

Published online: 8 May 2025

Abstract
Efforts to secure computing systems via software traditionally focus on operating system and application level remediations. In contrast, the Security Protocol and Data Model (SPDM) tackles firmware-level security challenges, which are much harder, if not impossible, to detect with regular protection software. SPDM includes key features such as enabling peripheral authentication, authenticated hardware measurements retrieval, and secure session establishment. Since SPDM is a relatively recent proposal, there currently exists a lack of studies evaluating its performance impact on real-world applications. In this article, we address this dearth by: (1) implementing the protocol on a simple virtual device, and then investigating the overhead introduced by each SDPM message; and (2) creating an SPDM-capable virtual hard drive and network card, and comparing the resulting read/write and transmit/receive performance against a regular, unsecured implementation. Our results suggest that SPDM bootstrap time takes in the order of tens of milliseconds to complete, while the toll of introducing SPDM on hard drives or network cards is highly dependent on specific workload patterns. For example, for mixed random read/write operations, the slowdown is negligible in comparison to the baseline unsecured setup. Conversely, for sequential read or write operations, the data encryption process becomes the bottleneck, reducing the performance indicators by several orders of magnitude

Keywords Security · Hardware · SPDM · Benchmark

1 Introduction

Modern computing devices are commonly built using several components from different manufacturers. This creates complex supply chains that usually end with an integrator, responsible for connecting off-the-shelf third-party components on an assembly line. After assembly, completed devices are tested before being shipped to warehouses, eventually reaching end users' facilities. Subsequently, the device components may undergo additional modifications, e.g. by means of firmware updates.

At every stage of the supply chain, there is a risk that malicious entities may inconspicuously tamper with or even entirely substitute components [1, 2]. The goals of such attacks may vary. For example, the attacks may aim to eavesdrop on sensitive data stored in volatile or non-volatile memory, or on information passing through video, audio, or network components. The attacks may also involve modifying default component behavior to enforce built-in obsolescence or impose some kind of censorship.

Although such attacks at the hardware/firmware level may seem to belong to the realm of fiction, the existence of successful proof-of-concept implementations indicates otherwise. For example, a recently published study describes a vulnerability in the remote firmware update functionality of a family of printers which allowed arbitrary code to be injected into the printer firmware [1]. The authors of the study then developed a self-replicating malware capable of eavesdropping on data and performing network reconnaissance. Another attack, this time targeting an USB firmware, enabled a device (e.g., a thumb drive) to impersonate a different device type (e.g., a keyboard) [3]. Beyond the halls of academia, there have also been real-world detections of firmware tampering attacks. In 2015, for example, hard drives

✉ Renan C. A. Alves
renanalves@usp.br

Bruno C. Albertini
balbertini@usp.br

Marcos A. Simplicio Jr.
msimplicio@usp.br

[1] Universidade de São Paulo, São Paulo, SP, Brazil

from various manufacturers had their firmware altered so that the modified code could be used to retrieve data even from encrypted partitions [4].

Protecting against such hardware-level tampering attacks can be a challenging task. This is particularly the case since the usual protection techniques that operate at the operating system level (e.g., antivirus software) or techniques that act at the infrastructure level (e.g., firewalls) are oblivious to such threats. Aiming to overcome this issue, the Security Protocol and Data Model (SPDM) [5] was proposed by the DMTF (Distributed Management Task Force) in 2019 to address low-level security challenges as the ones previously described. SPDM's main goals are to allow components to authenticate one another, to provide measurements of their internal state, and to securely exchange session keys. Firmware measurements enable system components to be verified, ensuring they have not fallen victim to tampering, while establishing session keys avoids passive eavesdropping from malicious components attempting to steal data. Considering the context of intrusion detection systems (IDS), the SDPM protocol could be used at the Physics-Based Monitoring level, considering the taxonomy proposed by Armellin et al. [6]. In addition to passive monitoring capabilities, SPDM enables active verification of firmware authenticity and measurements validity.

Albeit promising, SPDM is a relatively recent proposal, so its actual impacts on system performance have not yet been thoroughly evaluated in the literature. This article aims to close this gap by measuring the overhead added by SPDM's security layer, assessing how it could impact end-users' experience and identifying bottlenecks that might be optimized in the protocol. To the best of our knowledge, this is the first study to conduct a thorough performance analysis of the SPDM protocol. Specifically, we quantify the overhead added by the security layer to measure the impact on end-user experience using an emulated environment. It is important to note that we only focus on the performance aspect of the protocol. Any security-related tasks, such as attempting to retrieve session keys, forge signatures, or disrupt the protocol execution, are out of our scope. For a formal analysis of the SPDM protocol, we refer the reader to Cremers et al. [7].

The resulting contributions are twofold: (1) we evaluated the overhead of each individual phase of the SPDM protocol execution on a simple SPDM-enabled component, a random number generator (Sect. 5); and (2) we built two virtual SPDM-enabled devices: a hard drive and a network card. We assessed the impact on userspace-perceived performance (Sects. 6 and 7). All experiments build upon the virtualization capabilities of the QEMU emulator to implement our proof-of-concept devices and run the performance tests. For easy reproducibility, and also because the SPDM-enabled artifacts developed as part of this work may be of independent interest, the corresponding source code is publicly available at https://github.com/rcaalves/spdm-benchmark.

In summary, our results show that SPDM certificate-based bootstrapping procedure takes around 300 ms. Meanwhile, using SPDM to secure hard drive application data can greatly reduce the maximum transfer rate on sequential operations, but the impact was negligible on randomized mixed read/write workloads. Similarly, on the secured network card, the maximum data rate was also greatly reduced; conversely, transferring small files ($\leq 5kB$) was $\approx 10\%$ less efficient.

The rest of this manuscript is organized as follows. We discuss related work in Sect. 2. Section 3 summarizes SPDM's key aspects. In Sect. 4, we provide an outline of all experiments. In Sect. 5, we study the individual overhead of each SPDM message, describing also our methods and results. Section 6 contains the specification and results regarding our SPDM-enabled hard drive. Next, Sect. 7 describes our experiments concerning the SPDM-enabled network card. Finally, in Sect. 8, we present our closing remarks.

2 Related work

SPDM is a fairly new standard, so its impacts have not been thoroughly evaluated in the literature. Nevertheless, since SPDM's goal is to secure a system from early boot until OS runtime, it relates to other approaches that focus on the pre-OS stages. In particular, SPDM is somewhat similar to techniques often called "secure boot" or "trusted boot", whose goal is to ensure that only legitimate initialization firmware and bootloaders are executed.

To this end, the literature already includes a number of studies on secure boot performance. For example, Profentzas et al. [8] evaluate the overhead of software-based and hardware-based secure boot on embedded platforms, namely, Raspberry Pi and BeagleBone. Their study shows that the secured system presents an increased boot time ranging from 4 to 36%, depending on the techniques and algorithms employed.

In a similar fashion, Khalid et al. [9] proposed a trusted boot architecture for embedded systems based on an independent security processor. Their design was implemented on FPGAs, and their experiments show that their secured boot process increases boot time by 25%, in reference to a Linux system customized for their needs.

A study by Yin et al. [10] brought attention to failure-prone NAND flash chips, commonly used to store bootloaders in embedded platforms. The authors propose a redundancy scheme that verifies the integrity of bootloader firmware code and falls back to an alternative copy in the case of checksum mismatch. Their experiments show that total boot time is increased by 65% if the bootloader is intact. In the event of a

checksum mismatch, boot time is increased by 255%, since the backup code must be copied before being re-verified.

Kumar et al. [11] implemented a secure boot design based on post-quantum cryptography (PQC). Their main concern is that PQC algorithms require more computing resources than classic algorithms, which led them to create a custom FPGA implementation. Their experiments show that their implementation of the chosen PQC algorithm is at least 10 and at most 30 times slower than the baseline elliptic curve algorithm, depending on the level of parallelism.

Contrasting with the previous studies, the one by Dasari and Madipagda [12] is notable for being one of the few works in the current literature to include an analysis of SPDM. Specifically, the authors propose an architecture to detect component tampering in end products, producing a birth certificate comprising the platform as a whole. They use SPDM as part of their solution to retrieve firmware hashes (measurements) from the end product's individual components. However, their solution is based on an older version of SPDM, which did not yet support session key establishment. Consequently, their solution does not prevent passive sniffers from eavesdropping on sensitive information exchanged among components. At the same time, and contrary to this work, there is no evaluation of the impact brought by SPDM on application-level performance. Velozo et al. [13] explored the idea of re-purposing TLS fuzzers to analyze the SPDM protocol. They have selected a fuzzer among the investigated candidates, but have left the actual SPDM analysis for future works.

Ferreira et al. implemented a Wireshark dissector with the goal of facilitating the analysis of flowing SPDM messages [14]. This tool enables the visualization of SPDM messages stored in PCAP files.

When considering the particular scenario of protecting communications from/to hard disks, this current work shares a relationship with studies covering disk encryption technologies. Examples include works that focus on the energy consumption of full disk encryption technologies [15], on the impact of different cryptographic algorithms [16], or on specific types of devices [17–19]. There are also more holistic studies, such as [20], where architectures that include secure boot and hard drive encryption mechanisms are proposed (in this particular case, for mobile devices). Like the present study, such works give insights on how encryption impacts communications with hard disks, covering similar metrics. The similarity ends there, though. Since SPDM's secure sessions are meant to protect in-transit messages, not only the actual data written to or read from the disk, it cannot be used as an alternative to disk encryption, since all encrypted data sent to the disk is decrypted upon reception. Also, if disk encryption tools are employed together with SPDM, the protocol remains oblivious to the fact that it is protecting payloads already in encrypted form.

On the whole, we note that the aforementioned studies do not tackle the impact of SPDM, and its corresponding runtime security between components, at the application level. Therefore, their results are not directly comparable to the experiments presented herein. Previously, we presented a preliminary version of the secure hard drive [21] and the secure network card [22] in demo sessions. Given the limited publication space of these venues, we offer an extended examination of the results herein, namely, we have updated SPDM to version 1.3, provided implementation details, added new experiments, and expanded the discussion of the results.

3 SPDM

This section summarizes basic concepts and the workflow of SPDM [5].

SPDM is a standard for secure intercommunication among hardware components. It follows a requester-responder paradigm, and focuses on defining a set of useful operations and message formats that enable mutual authentication and the establishment of secure channels over an insecure medium. At the same time, it aims to be agnostic to the physical medium and the encapsulation approach employed for conveying those messages.

A subset of SPDM messages is related to core functions, such as device authentication, measurement retrieval, and secure session establishment. The other messages serve the purpose of reporting available resources, stating which optional features are present, negotiating the cryptographic algorithms to be employed, and maintaining a communication session as active. Figure 1 gives an overview of the expected message flow, highlighting which messages are not mandatory. Since the main messages are evaluated in our experiments, we briefly describe them in the passages that follow.

The first pair of messages are GET_VERSION and VERSION. They are employed to settle on the SPDM version to be used. The protocol proceeds only if at least one of the versions advertised by the responder is supported by the requester. Next, the requester inquires the responder about its capabilities, aiming to discover which optional messages are supported (messages GET_CAPABILITIES and CAPABILITIES). The last pair of mandatory messages is NEGOTIATE_ALGORITHMS and ALGORITHMS. They are exchanged so the requester and the responder can agree on the set of cryptographic algorithms they will use henceforth, throughout the protocol execution. These mandatory messages are expected to present low overhead, since they have a slim payload and their processing does not involve any compute-intensive operations.

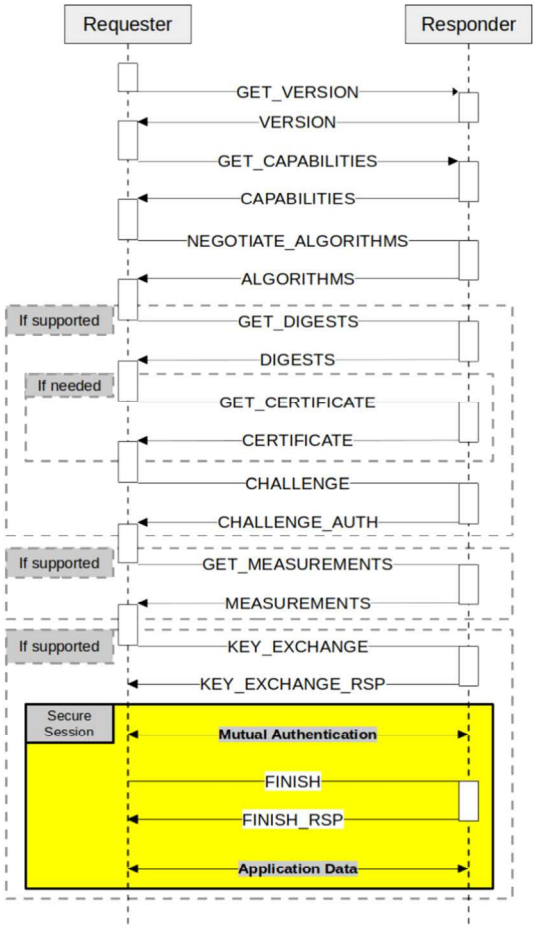

Fig. 1 SPDM message flow. Adapted from [5]

The next set of messages serves the purpose of retrieving certificates. The GET_DIGEST/DIGEST message pair enables the requester to check whether any of the responder certificates have been previously fetched and cached. If that is not the case, certificates are retrieved via the GET_CERTIFICATE /CERTIFICATE message pair. After obtaining the responder's certificate, the requester may challenge the responder to prove that it is the rightful owner of the corresponding public/private keys through the CHALLENGE/CHALLENG_AUTH message pair.

A measurement may represent firmware, software, or configuration data of an endpoint that helps to ensure that it is not counterfeit. The requester sends GET_MEASUREMENTS message to request measurements, which is answered by a MEASUREMENTS message. The requester usually demands the responder to sign MEASUREMENTS messages, although such a signature may also be omitted.

The requester can also issue a KEY_EXCHANGE message aiming to initiate a secure handshake for establishing a shared secret key. The responder then answers with a KEY_EXCHANGE_RSP message. The handshake is finalized by a FINISH/FINISH_RSP message pair, when the secret key computed by both endpoints is confirmed and bound to the corresponding secure session established between them. Alternatively, if a pre-shared key is used instead of certificates, the messages above are replaced by their PSK_ counterparts, i.e., PSK_KEY_EXCHANGE, PSK_KEY_EXCHANGE_RSP, PSK_FINISH, and PSK_FINISH_RSP.

After a session is established, the underlying keys can be updated with the KEY_UPDATE and KEY_UPDATE_ACK messages. Also, if a session timeout period has been configured, HEARTBEAT/HEARTBEAT_ACK messages can be used to keep the session alive even in the absence of regular traffic. Sessions can be terminated by means of the END_SESSION/END_SESSION_ACK messages.

Finally, exceptions during the protocol execution can be flagged by ERROR messages, such as INVALIDREQUEST, BUSY, and DECRYPTERROR.

4 Experiment goals

Our goal is to estimate the computational cost of employing SPDM to provide low-level device authentication and low-level hardware communication confidentiality. This was achieved by comparing the performance of SPDM-enabled devices with their plain counterparts. Some of the experiments are designed to estimate how user-perceived metrics are affected.

The experiments are divided in three sections, depending on the type of device that was converted to be SPDM-enabled. Each section is further divided into subsections containing implementation details, experimental method, and results.

First, we designed a simple virtual device that generates random numbers (Sect. 5). We use this device to measure the cost of each SPDM message individually. The obtained results can be used to estimate the additional CPU load given a sequence of SPDM messages.

Next, we analyze the hard drive use case in Sect. 6. Since hard drive performance is known to be heavily influence by rotation and seek times, we used different tools to test a variety of workloads.

Finally, Sect. 7 presents the network card use case. A network card was picked since network cards typically have no clear performance bottlenecks, unlike hard drives. We tested the behavior of different workloads, comparing TCP and UDP protocols in the light of throughput and latency metrics.

5 SPDM overhead assessment

The goal of this experiment is to assess the overhead introduced by each phase of the SPDM message flow. To this end, we developed a random number generator (RNG) device that

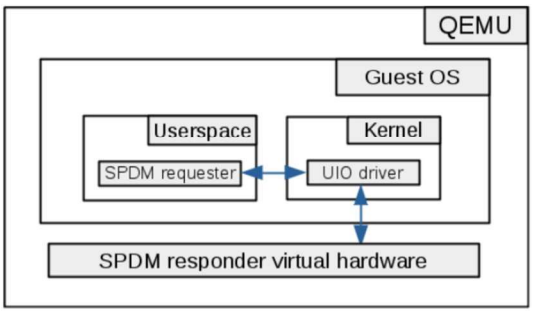

Fig. 2 System architecture

supports two operation modes: 1) secure mode, which uses SPDM to enable authentication and secure session establishment; and 2) clear-text mode, devoid of any SPDM-related security capabilities.

5.1 Method

We used a virtualized setup based on the QEMU emulation software [23]. The virtual machine run by QEMU contains an instance of our RNG device, which implements an SPDM responder.

The RNG was designed as a PCI device with a memory-mapped input/output (MMIO) region to send and receive SPDM messages, as well as to exchange control information. Non-SPDM transactions were used as a baseline for estimating the overhead introduced by SPDM to this simple procedure.

We implemented an SPDM requester as a device driver for our RNG on Linux. This device driver is built upon the userspace I/O system (UIO). Figure 2 depicts a schematic of our setup. Other implementation details are presented in Appendix A.

In our experiments, the interaction between the RNG device and its device driver was conducted following five steps: 1) SPDM library initialization; 2) SPDM connection, comprising version, capabilities, and algorithms negotiation; 3) SPDM authentication, including digest, certificate, and challenge messages; 4) measurement retrieval; and 5) application phase.

We analyzed the measurement messages in two ways: 1) retrieving each measurement individually; and 2) retrieving all of them at once. In the first scenario, the requester first inquires about the total number of measurements the responder holds before retrieving each one individually, while requiring a signature only for the last one. In either scenario, the responder was configured to hold 5 different measurements, following the DMTF measurement block specification (SPDM specification v1.3.0 [5], Section 10.11.1.1). Each block contains 128 bytes of dummy data.

Table 1 `libspdm` parameters

Parameter	Value
SPDM protocol version	1.3
`libspdm` version	3.2.0
`libspdm` build options	x64, release
Underlying crypto library	MbedTLS
Requester signature algorithm	RSAPSS_3072
Responder signature algorithm	ECC_NIST_P384
Measurement hash algorithm	SHA384
DHE algorithm	SECP_384_R1
AEAD algorithm	AES_256_GCM
Key scheduling algorithm	As defined by SDPM
Mutual authentication	enabled

The application phase is also divided into five steps: 1) the endpoints agree on a shared key; 2) a heartbeat message is transmitted; 3) the session key is updated; 4) the requester asks for a random number from the responder; and 5) the session is concluded. The session establishment process was tested both in the pre-shared key (PSK) and in the certificate-based settings.

We used the kernel's performance event infrastructure API to extract performance indicators (`perf`) [24], allowing us to assess the cost of each message individually. We focused on two metrics provided by the `perf` API: number of cycles, and total CPU time. CPU time provides a tangible sense of how long a task takes to finish. However, it depends on the underlying hardware. The number of cycles, on the other hand, is a more generic metric in comparison with CPU clock time measurements, although it remains platform dependent. Furthermore, we configured `perf` to exclude from the count any event that occurs in kernel space, at the hypervisor, or at the guest machine (for responder only).

SPDM functionality was provided by the `libspdm` open source library [25]. Compile-time and execution-time parameters used in the experiment are listed in Table 1.

As the emulation environment, we used QEMU version 4.1 [23]. A QEMU virtual machine is specified by command-line options: hard drives, CPU, network cards, along with other options. The parameters used in the experiments are listed in Appendix A. The guest operating system was obtained from Buildroot[1] version 2020.02.9, which contains Linux Kernel version 4.19.

The host system ran a Linux-based system on an Intel i7-10700KF processor, with 3800 MHz clock, 8 Cores, 16 logical processors, and 32 GB of RAM. Although real-world devices running the SPDM protocol are likely to have lower processing power, the results obtained should

[1] Available at https://buildroot.org/.

be proportionally valid and, hence, useful for our comparison purposes. More precisely, running SPDM on a slower CPU following a similar architecture is expected to result in a similar number of cycles. Conversely, the measured execution time should increase proportionally on slower CPUs in both SPDM and non-SPDM scenarios. Ideally, the CPU clock should be constant while performing benchmarks. Hence, aiming to approach this ideal scenario, the following options were disabled in the machine BIOS configuration menu: turbo boost, speed step, speed shift, hyper-threading, and CPU C states. In spite of disabling these options, our results still displayed varying clock frequencies. In particular, requester-side operations require waiting for the requester to respond leading to lower clock frequencies, while long CPU-intensive operations tend to present higher clock frequencies.

Each of the execution steps was performed 100 times, aiming to obtain statistical confidence. The graphs shown in Sects. 5.2 and 5.3 present the average along with 95% confidence intervals.

5.2 Results: requester

Requester results are presented in Fig. 3, for both cycle count and execution time. As expected, most of the overhead was due to messages related to the authentication process.

Figure 3 shows that the most time-consuming messages are `GetCertificate`, `KeyExchange`, and `Finish`, taking respectively 16.4, 24.9 and 8.9 ms, or 16.8, 62.1 and 29.4 million cycles on average. The `GetCertificate` procedure is expected to be slow since 1) it may require several messages to complete, and 2) by the end of it, the retrieved certificate must be verified for correctness, which requires a few time-consuming signature verifications. `KeyExchange` and `Finish`, in turn, involve the generation of a symmetric key pair by means of a Diffie-Hellman key exchange, which is computationally expensive.

On the other hand, the usage of Pre-Shared Keys (PSK) considerably reduces the burden of establishing session keys. The precise difference is that `KeyExchangePSK` takes only 1.2 ms (7.0×10^5 cycles), which is only a fraction of the 24.9 ms (6.2×10^7 cycles) observed in its asymmetric counterpart. Also, if the ultimate goal is to establish a secure session, using a PSK setting allows components to forego `GetCertificate` and `Challenge` messages, further speeding up the key establishment process.

Other time-consuming messages are `GetMeasurements` and `Challenge`, both of which have an execution time of around 3 ms. Interestingly, we noticed that retrieving measurements all at once is faster than retrieving measurements one by one. Specifically, the time taken in the former case is 3.1 ms, against 5.2 ms in the latter. This represents a time gain of 40%, while the number of cycles is reduced by 46%. This discrepancy stems from the additional overhead of exchanging several short messages instead of a single one.

Each of the other messages takes from 260 μs up to 1.4 ms on average, depending on their underlying complexity, so they are unlikely to become bottlenecks in practice. In particular, the three most basic messages (`GetVersion`, `GetCapabilities`, and `NegotiateAlgorithms`) take 1.1 ms combined to execute ($3.2E+05$ cycles). Conversely, we notice that the task of loading the certificate from the disk can take a significant amount of time: 37.3 ms, or 1.3×10^8 cycles, on average.

Once a secure session is established, we were able to retrieve a random number from the SPDM-enabled RNG device in 464 μs (1.5×10^5 cycles). Conversely, in our SPDM-free baseline execution the same operation took 74 μs (5.9×10^4 cycles) on average. This means that SPDM led to a 5.2-fold increase in terms of time, or a 24-fold increase in number of cycles. This result is unsurprising, though, when we take into account the remarkable simplicity of the device evaluated in our tests. Our simple RNG was designed to be extremely fast, so even cryptographic operations which are quite lightweight in absolute numbers, such as symmetric encryption, become comparatively expensive in the grand total of each test.

5.3 Results: responder

Figure 4 shows the metrics extracted from the Responder. As shown by this figure, `KeyExchange` is the most expensive message to process, taking 109 ms, or 4.1×10^8 cycles. As expected, though, adopting a PSK setting reduces the session key establishment overhead from 124 ms to only 408 μs or 1.3×10^6 cycles.

The responder handles `GetCertificate` messages faster than the requester. The reason behind this behavior is that most of the cryptographic processing of `GetCertificate` remains at the requester. At the responder, `GetCertificate` takes only 144 μs, or 4.9×10^5 cycles.

Similarly to the requester, `GetMeasurements` and `Challenge` are nearly tied as the second most time-consuming operations at the responder, taking approximately 250 million cycles or 67.5 ms. Once again, retrieving measurements all at once was faster than retrieving one by one, however, the obtained values are statistically equal.

The responder was fast at loading certificates from the disk, taking only 3.8 ms or 1.3×10^7 cycles (compared to 37.3 ms at the requester). The discrepancy between responder and requester is caused essentially by our configuration, where each party uses a different signature algorithm. More precisely, the responder uses a signature algorithm based on elliptic curves, which takes less time to verify than the RSA-based signatures generated by the requester.

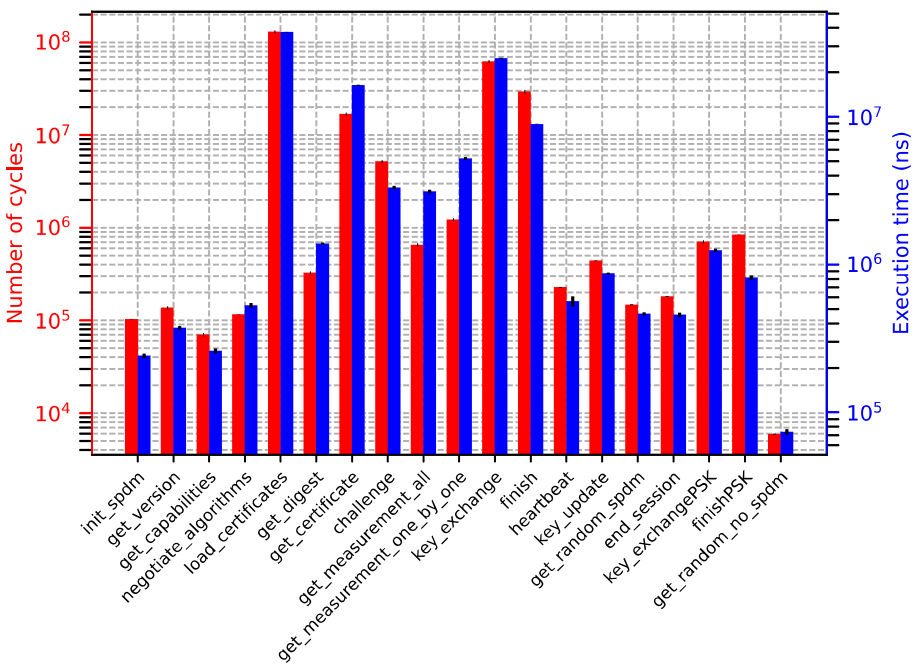

Fig. 3 Requester execution time and number of cycles

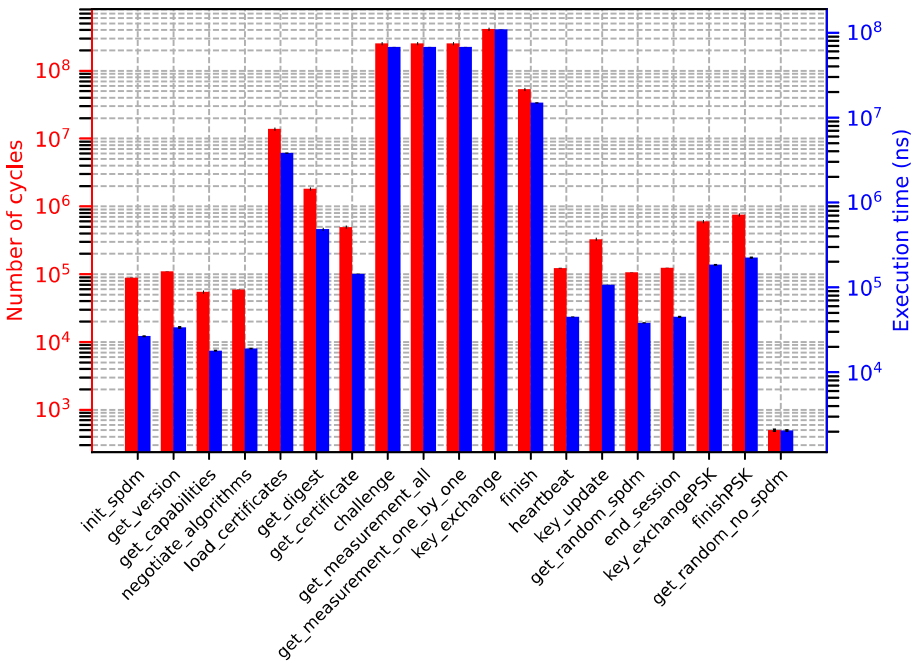

Fig. 4 Responder execution time and number of cycles

Processing a random number request without SPDM took the responder 2.1 μs, or 499 cycles. Adding the SDPM layer increases this cost to 38.1 μs, or 1.1×10^5 cycles. The processing of any other messages took from 18 to 107 μs (or 5.4×10^4 to 3.3×10^5 cycles), adding a moderate amount of overhead to the protocol.

6 Hard drive use case

This experiment was designed to assess the impact SPDM poses on system performance from a user perspective. Among the possible peripherals typically found in computing systems, we chose to secure the communication between CPU and hard drive, both due to this component's importance and to the availability of tools for conducting such performance tests. Specifically, we compared an SPDM-equipped

hard drive against an unsecured one under various workloads, considering boot time, read speed, and write speed as metrics.

6.1 Implementation details

Due to the lack of off-the-shelf SPDM-enabled hardware, we once again resorted to emulation. We used QEMU as the emulation software, since it is equipped with a variety of open source virtual devices, including hard drives. After evaluating the available options, we chose to work with the `VirtIO` hard drive. Both Linux driver and virtual hard disk were modified to incorporate SPDM security functionalities, provided by `libspdm` (using the same parameters listed in Sect. 5.1). Implementation details are described in Appendix B.

6.2 Method

In our experiments, we used a few widely employed tools and benchmarking utilities to assess hard drive performance: `dd`[2], `hdparm`[3], `ioping`[4], `bonnie++`[5], `fio`[6]. These tools and their configuration are further detailed in Appendix D. All tests were executed on a separately attached hard drive. Once again, the following CPU attributes were disabled at the machine BIOS configuration menu: turbo boost, speed step, speed shift, hyper threading, and CPU C states. For each tool, two batches of experiments were executed: 1) a baseline setting, with the unmodified device and driver; and 2) an SPDM-enabled setting, with our modified implementations.

6.3 Experimental results

This section presents and discusses the results obtained from each of the benchmark tools used.

6.3.1 `dd`

This is a commonplace utility found on Unix-like operating systems. Its primary use is to transfer raw data from a source to a destination.

We tested writing 2GB of data to the disk according to two approaches: in small blocks of 4KB each, or in 512MB-long blocks.

Figure 5 shows the results obtained from the `dd` command. This experiment assesses the speed of writing data sequentially, which reduces the number of seek operations executed during the test. Hence, besides providing insights into SPDM's impact on disk writing speed when such workloads are prevalent, these results also serve as a baseline for scenarios where random disk accesses are more common.

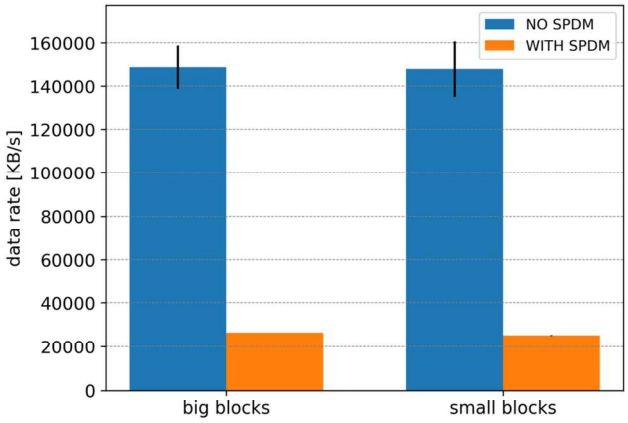

Fig. 5 Measuring data rate of an SPDM-enabled hard disk: `dd`

In general, writing small blocks tends to be slightly slower than writing large blocks, which was observed in our results (≈ 1% slower without SPDM, ≈ 5% slower otherwise). However, in both cases, SPDM caused a ≈ 82% slowdown in writing speed.

6.3.2 `hdparm`

The `hdparm` tool provides an indication of sequential data read speed, disregarding filesystem overhead. Without SPDM, the average read speed observed was 3.8 GB/s, fairly close to the nominal 6 GB/s speed of the hard drive, considering the virtualization overhead. However, introducing SPDM drastically decreases the value indicated by hdparm to 20 kB/s, translating to a 99.5% speed degradation.

6.3.3 `ioping`

This is a tool for monitoring disk latency. Figure 6 shows read and write latency results according to `ioping`. Focusing on average results only, the introduction of SPDM reduced the reading and writing latency by 20% and 14%, respectively. However, the obtained confidence intervals in both cases were very large, making it difficult to draw statistically relevant conclusions with this tool. All in all, these results hint that latency was not affected by the introduction of SPDM.

6.3.4 `bonnie++`

This is a purpose-built benchmarking toolkit for hard drives. The results (Fig. 7) indicate a large loss of performance when SPDM is introduced. The tests performed by this tool consist of reading and writing 200 MB files to the disk until amounting 2GB. The performance proportion is somewhat similar

[2] https://man7.org/linux/man-pages/man1/dd.1.html
[3] https://sourceforge.net/projects/hdparm/
[4] https://github.com/koct9i/ioping
[5] https://doc.coker.com.au/projects/bonnie/
[6] https://fio.readthedocs.io

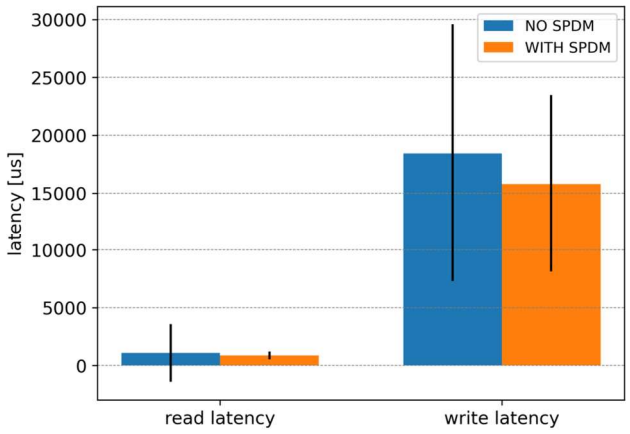

Fig. 6 Measuring latency of an SPDM-enabled hard disk: `ioping`

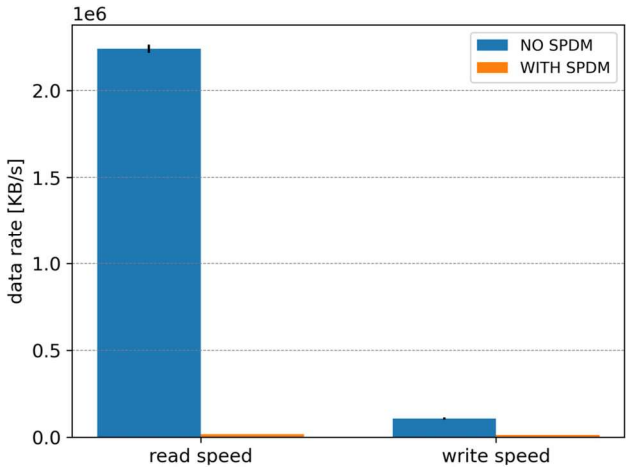

Fig. 7 Measuring disk datarate: `Bonnie++`

to the one obtained by the `dd` writing test, but spread across multiple files. However, the numbers show a deeper performance chasm than what was observed with `dd`: the writing speed drops from 108 to 13 MB/s (an 88.1% reduction). The loss of reading performance is even more significant: from 2.2 GB/s to only 17 MB/s. It makes sense that both reading and writing speeds drop by the same order of magnitude since the system's bottleneck is the same in both tests – the processing cost of encrypting/decrypting every transaction.

6.3.5 `fio`

This is a highly customizable benchmarking tool for hard drives that aims to create workloads as close as possible to the desired test case.

Results from the `fio` tool are presented in Fig. 8, where the unit of measure is input/output operations per second (iops). The sequential tests (i.e., those labeled 'sequential read' and 'sequential write') consider large blocks while requesting disk synchronization sparsely. In this case, the

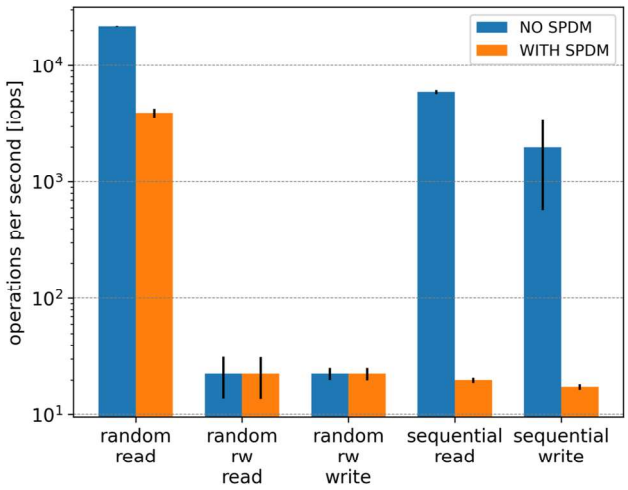

Fig. 8 Measuring transaction rate of an SPDM-enabled hard disk: `fio`

pattern observed is similar to the ones obtained with `dd` and `bonnie++`: there is a significant loss of performance, with the number of iops in the SPDM-enabled disk being less than 1% of the baseline value.

The performance degradation becomes less prominent when randomness is introduced. For these tests, the block size was set to 4kB, and a request to synchronize data was sent after every operation. When performing random read-only tests, the performance drops to 18.2% of the baseline value. Mixing randomized reading and random writing operations (in Fig. 8, label "random rw read" refers to the read speed, and label "random rw write" refers to the write speed) yields virtually the same level of iops; introducing SPDM causes a reduction of approximately 0.5% in both cases, but the standard deviation width prevents us from attesting statistical difference. We conjecture that the reason behind this trend is the bottleneck shifting from the cryptographic operations to the physical magnetic disk operations, namely the frequent seek operations to address the random request locations.

6.3.6 Boot time

During OS initialization, the SPDM-enabled HD driver performs all SPDM bootstrapping procedures, including loading certificates and establishing a symmetric key. Our test shows that these procedures increase OS boot time by 205ms (1.06 s to 1.265 s), which resonates with the results presented in Sect. 5.

6.3.7 Summary

In summary, the results show that bulk operations are greatly affected by securing the hard drive's low-level communi-

cations with SPDM. However, disk access patterns that mix small read and write operations performed nearly at the same speed when compared to a plain baseline.

7 Network card use case

The experiment introduced in this section aims to assess the impact of SDPM on another device class: network cards. Although throughput and latency are still the main metrics of interest, workload and impact on user experience are fundamentally different in these scenarios compared to the hard drive study presented in Sect. 6. The influence of SDPM is studied by comparing the outcome of benchmark tools between an SPDM-equipped card and its original counterpart. Implementation details are described in Appendix C.

7.1 Method

All tests were executed on a QEMU virtual machine equipped with an E1000 network card.

We used two benchmarking utilities to assess networking performance: `iperf3`[7], and `netperf`[8]. A more detailed description of these tools is provided in Appendix E. Additionally, we set up web servers within both the host and the guest to check how long it takes to retrieve files of different sizes. For each case, two batches of experiments were executed: 1) a baseline setting, with the unmodified device and driver; and 2) an SPDM-enabled setting, with our modified implementation.

7.2 Results

This section presents and discusses the metric values obtained by each of the benchmark methods, with the goal of characterizing the impact introduced by SPDM. The graphs are labeled from the guest machine point of view, that is, “send" means packets were sent from the guest machine to the host machine, whereas “receive” means packets were sent from the host machine to the guest machine.

7.2.1 `iperf3`

Throughput, one of the main network metrics, is shown in Fig. 9. As may be expected, introducing SPDM causes an overall reduction in throughput, since every packet is encrypted at a low level. This throughput reduction ranged from 23% (UDP receive) to 84% (TCP send).

Next to throughput, delay is another key network performance indicator, shown in Figure 10. The `iperf3` tool only records this metric for `TCP` tests. Again, the SPDM-secured system performs worse, increasing the round trip time from 83 μs to nearly 385 μs for traffic flowing from the VM to the host (TCP send). Although it is nearly a 5x increase, it is imperative to remember that both server and client are running on the same machine. Therefore, many common delay sources are not factored into the measurement, such as propagation delay and queuing delay on routers. Considering typical delay over the internet is in the order of milliseconds, one could argue that extra 0.3 ms would negligibly impact most applications.

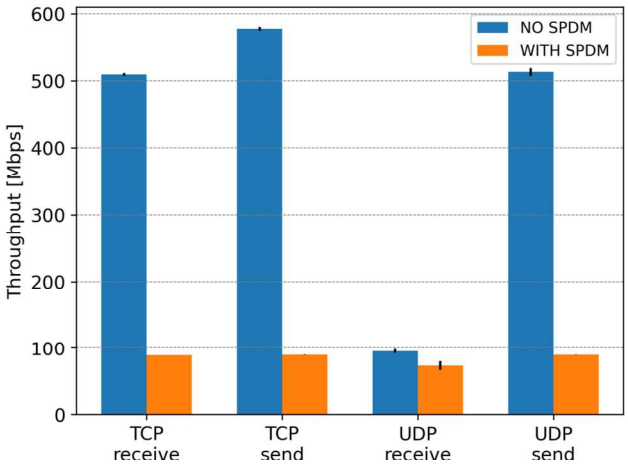

Fig. 9 Measuring network throughput: `iperf3`

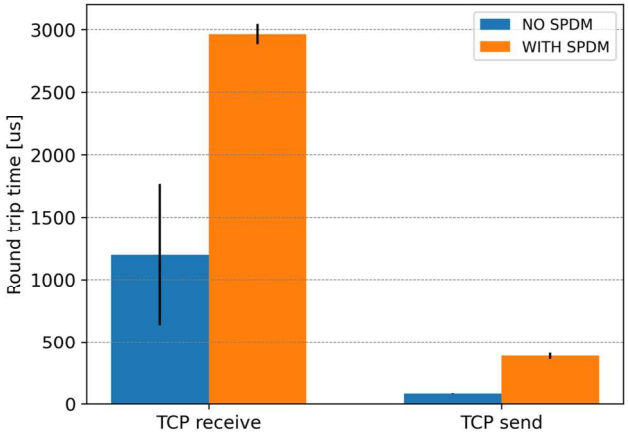

Fig. 10 Measuring round trip time: `iperf3`

Traffic in the reverse direction presented a larger delay: 1.2 ms without SPDM increasing to nearly 3 ms on the secured version. The extra 1.8 ms is still tolerable in most applications, although it could impact delay-sensitive applications, for example, applications requiring haptic feedback.

The jitter metric (delay variation) is collected by `iperf3` only for UDP tests (Fig. 11). The results indicate that introducing SPDM tends to increase the jitter (between 1.3x and

[7] https://iperf.fr

[8] https://github.com/HewlettPackard/netperf

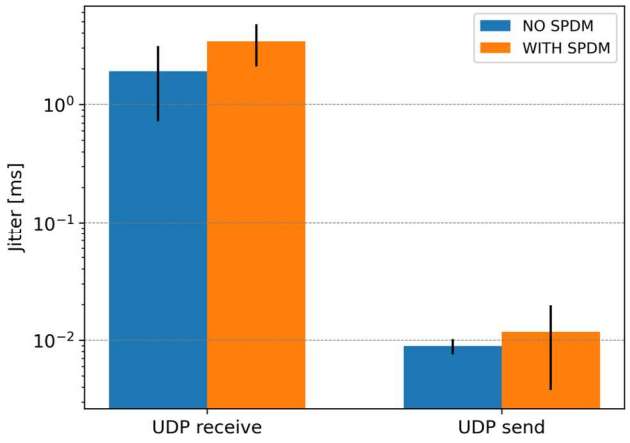

Fig. 11 Measuring network jitter: `iperf3`

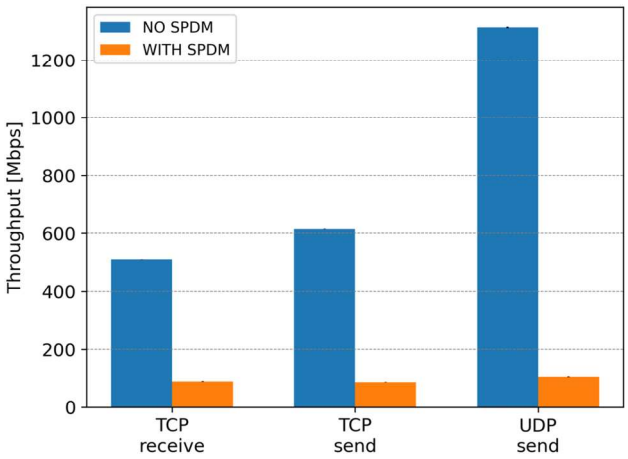

Fig. 12 Measuring throughput [MBps]: `netperf`

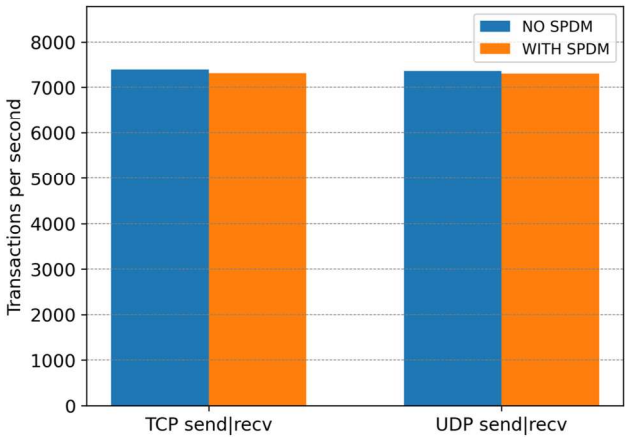

Fig. 13 Measuring throughput [T/s]: `netperf`

1.8x), although the wide confidence intervals prevent us from drawing any definitive conclusions.

Another metric collected only for UDP traffic is loss rate (since UDP is a best effort protocol, whereas TCP provides reliability). Considering the `UDP send` scenario, there were no packets lost. However, during the `UDP receive` scenario, massive losses were recorded: approximately 98%.

These loss rate results are linked to the CPU usage. On `UDP send`, the transmission rate is bound by the guest machine's speed, which is slower than that of the server, resulting in all packets being processed accordingly. However, in the `UDP receive` scenario, the host machine transmits packets as fast as it can, completely overwhelming the guest machine.

According to `iperf3`, considering UDP traffic, the packet sender achieves nearly 100% CPU usage, since it continuously sends UDP packets as fast as it can; whereas the receiver presented CPU usage between 4% and 20%. Curiously, in the `UDP send` scenario, the packet receiver displays a higher CPU usage without SPDM since it processes more packets (as indicated by throughput in Fig. 9).

The SDPM overhead dominates the guest machine's CPU usage in both both `TCP send` and `TCP receive`, increasing to nearly 100% and 60%, respectively. On the host machine, the CPU is not a bottleneck, with the larger number of processed packets leading to a higher CPU usage in the scenario without SPDM.

7.2.2 `netperf`

The throughput results obtained from netperf are displayed in Figs. 12 and 13. The first set of results represents a greedy sender benchmark, measured in MB/s: one of the endpoints sends as much traffic as it can. Thus, the overhead introduced by SPDM is detrimental to performance, presenting between 83% (TCP receive) and a 92-fold (UDP send) smaller maximum datarate, similar to what was observed with `iperf3`.

In the second set of results, the benchmark executes synchronous request/response transactions, one at a time. In addition, the transactions payload consists of only 1 byte of dummy data. These two characteristics combine to reduce the strain on the CPU since less data is transmitted (and thus encrypted). The results show that, under these conditions, the overhead introduced by securing the messages at hardware level is barely noticeable (less than 1.1% degradation).

In short, the latency metric yields proportional results in comparison to throughput: the observed delay is a few times larger with SPDM (6x to 12x), except for the transaction-based test cases. Nonetheless, the largest delay measured was approximately 5 ms, which is acceptable for most applications (Fig. 14).

The CPU load on the guest machine is nearly 100% in TCP send and UDP send scenarios (greedy tests), regardless of the presence of SPDM. In the TCP receive scenario, using SPDM caused the CPU load to increase by 81%, while the

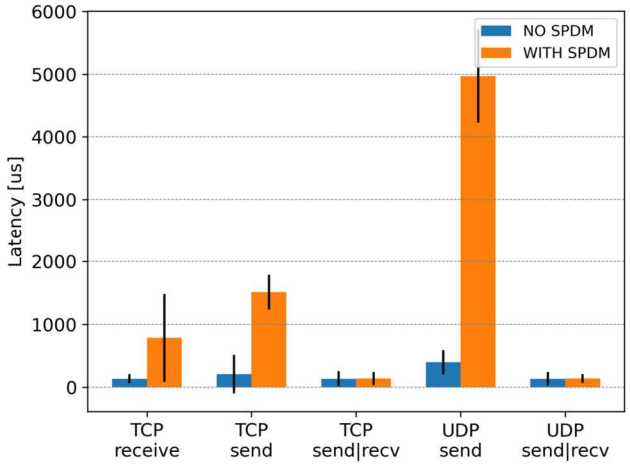

Fig. 14 Measuring network latency: `netperf`

increment in transaction-based scenarios was approximately 36%.

The CPU usage of the host machine did not seem to be greatly impacted by SPDM. Although the tests with SPDM tended to present higher CPU utilization, this was not observed in every scenario.

7.2.3 Download from web server

Lastly, the results from the file download test are shown in Fig. 15, containing the average data rate.

For files ranging from 5kB to 500kB, the performance gap between the secured and unsecured implementations increased steadily. This trend can be explained by the presence of overheads other than the hardware level SPDM encryption that are proportionally large, such as TCP windowing, interrupt handling, disk I/O operations, and OS packet queue management.

For files larger than 500kB onward, the difference stabilizes: the secured version takes around 5 times longer to complete the download ($\approx$ 80% lower throughput). The SPDM-secured implementation peaks at 10.5 MB/s, while the original implementation peaks at 60.5 MB/s.

7.3 Summary

Adding SPDM to the network card and encrypting all application-related payloads can severely reduce the maximum achievable throughput. Nonetheless, for use cases that are not throughput-intensive presented manageable performance losses. A comprehensive list of all numerical results is in Appendix F. We speculate that, in future SPDM implementations, dedicated cryptographic hardware may reduce the performance strain imposed by the additional security layer.

8 Final remarks

The Security Protocol and Data Model (SPDM) aims at providing standardized ways for component (mutual) authentication, firmware integrity check, and secure communication establishment.

Although these functionalities are important for increasing the security of modern computing systems, they are expected to cause tradeoff performance penalties. Our goal in this paper is to assess the magnitude of SPDM's performance impact.

In general, our results show that the overhead introduced by the most time-consuming SPDM message is 109 ms (409 million cycles), while the fastest messages take only a few microseconds. According to our experiments, the typical SPDM bootstrap takes approximately 300 ms to run, considering both requester and responder typical operations.

Regarding the hard drive benchmarks, we note that the specific workload greatly influences the final outcomes. On sequential read or write operations, data encryption becomes the bottleneck, and heavily affects performance (e.g., reading speed dropped from 2.1 to 17 MB/s in one run of the benchmark). That is not the case, however, for workloads that are mainly comprised of random read and write operations scattered throughout the disk. For such workloads, we found no significant performance differences between the secured system and the baseline system. The bottleneck then becomes the physical movement of disk heads and switching between reading and writing modes.

Lastly, the experiments on the SPDM-secured network card revealed that the maximum throughput can be severely reduced, the worst case being a reduction from $\approx$ 1300 to $\approx$ 130 MB/s of UDP traffic. However, this performance loss was partially due to the reduced processing power of the guest machine. This result highlights that the processing imposed by the additional security layer is especially burdensome to embedded systems. Other experiments showed that, if only small amounts of data are transmitted, the performance drop remains contained (1%, at most).

In future work, we intend to further explore the performance impacts of deploying SPDM on modern systems. This includes conducting experiments on physical hardware platforms instead of using emulation software, which can be achieved by, for example, implementing SPDM-enabled devices on FPGAs.

Appendix A Random number generator (RNG) implementation details

As discussed in the main document, the RNG device was implemented as a memory-mapped PCI device. After initializing, the RNG waits until the device driver writes a

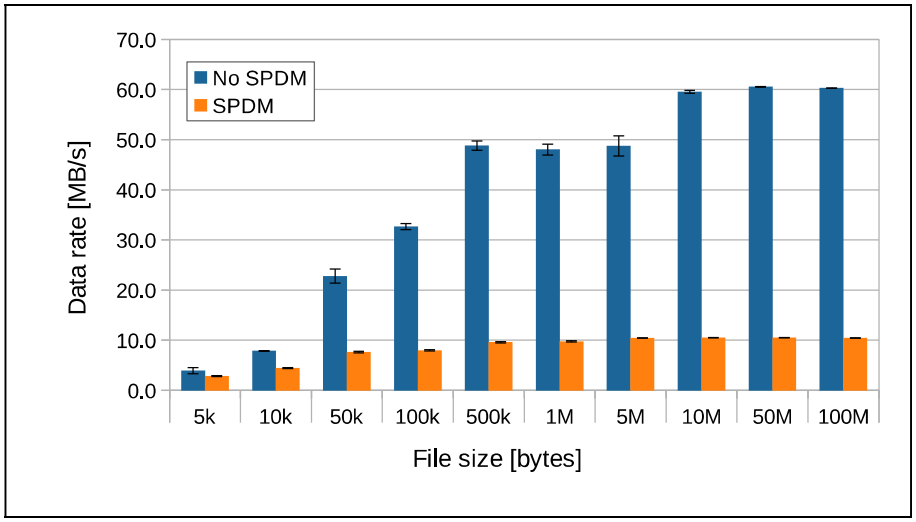

Fig. 15 File download test: data rate

request message to a chosen address in the MMIO region, and then indicates that the message is ready by writing to a control register. The message can only be read after the device driver writes to the control register indicating it is ready. After reading and processing the message, the RNG device sends an interrupt signal to announce that the response can be read from the MMIO region. Some of the memory region addresses are destined for non-SPDM transactions, from which the baseline results for retrieving a random number outside of an SPDM secure session were extracted.

We implemented the SPDM requester as a device driver for our RNG device on Linux. This device driver is built upon the userspace I/O system (UIO).

For better reproducibility, the QEMU command-line options used in the experiment are listed below:

- `-enable-kvm`: enables KVM (Kernel-based Virtual Machine), which enhances virtual machine performance. Needed so the guest kernel's performance event infrastructure is granted access to hardware counters;
- `-cpu qemu64,pmu=on`: selects the emulated CPU model and enables Performance Monitoring Unit (PMU), needed to access performance counters;
- `-device spdm`: attaches an SPDM RNG device to the virtual machine;
- `-kernel bzImage`: selects the kernel booted by the virtual machine. We used Linux Kernel version 4.19 from Buildroot;
- `-drive file=rootfs.ext2,if=ide,format=raw`: indicates the root file system used in the VM, using IDE interface, and raw format. We used the root file system from Buildroot;
- `-append "console=ttyS0 rootwait root=/dev/sda"`: kernel command-line options. Sets the default console output, waits until root device is ready, and sets the root file system partition.
- `-m 1024`: sets the amount of RAM at the virtual machine, in megabytes,

Appendix B Hard drive implementation details

The guest operating system is a custom Buildroot-based Linux distribution[9] Its kernel contains a native `virtio_blk` driver, compatible with QEMU's `virtio_blk` hard disk. Both Linux driver and virtual hard disk were modified to incorporate SPDM security functionalities, provided by `libspdm`.

In short, the interaction between the kernel driver and the virtual device is as follows: 1) the operating system sends read/write requests[10] to the driver queue (`queue_request` function); 2) the request is executed and transferred from the guest virtual machine's kernel space to the virtual disk's request handler, triggering the `handle_request` function; 3) the request is forwarded to the host's disk; 4) QEMU receives the request results from the host OS (`rw_complete` callback function); 5) QEMU forwards the results to the `virtio_blk` driver, activating the `request_done` function; 6) the guest kernel is informed that the I/O operation is complete. The diagram in Fig. 16 illustrates these aforementioned steps.

The aforementioned interaction was adapted to follow the SPDM workflow.

In the adaptation, the driver fills the role of the SPDM requester, while the hard drive takes the role of SPDM responder. Requester and responder are required to bootstrap the

[9] Available at https://buildroot.org/.

[10] There are other kinds of operations, but we focus on reading and writing for illustrative purposes.

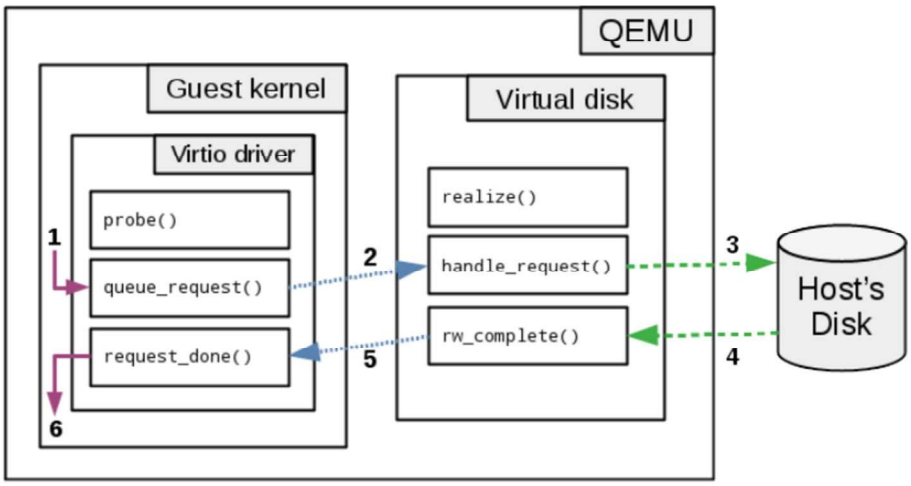

Fig. 16 `virtio_blk` driver and virtual hardware interaction: 1) Guest operating system request, 2) Driver request to virtual hardware, 3) Request forwarded to host, 4) Request result at virtual hardware, 5) Request result at driver, 6) Request finished

SPDM protocol before heading into the application phase, in which read/write requests are encrypted.

The `virtio_blk` hard drive initializes internal SPDM variables and loads certificates during the virtual machine initialization (in the `realize` function). After calling this function, the hard drive is ready to process incoming SPDM messages.

The kernel driver performs a similar initialization when a new `virtio_blk` device is detected, using the `probe` function. At that point, not only are local variables initialized, but the whole SPDM bootstrap procedure also occurs. As a result, all SPDM messages, from GET_VERSION to KEY_EXCHANGE, are exchanged at that time. All SPDM messages are encoded similarly to regular read/write requests, but using a special operation code. By the end of `probe` function, driver and hard disk obtain a symmetric key they can use for encrypting application data.

Incoming write requests are now encrypted as part of the `queue_request` function in the kernel driver, and decrypted at the `handle_request` function when it reaches the virtual disk. Analogously, a read request is encrypted after the data is retrieved from the host by means of the `rw_complete` function (on QEMU), and decrypted by the kernel driver as part of the `request_done` function.

Appendix C Network card implementation details

The experiments were conducted on a QEMU emulated environment. From the available virtual network card implementations, we chose to work with the E1000 card, since its implementation strikes a good balance between functionality, simplicity, and documentation availability. By default, our custom Buildroot-based Linux distribution contains a compatible device driver. Both the driver and virtual device were modified to incorporate SPDM functionalities.

In brief, the guest OS sends packets as follows: 1) on the kernel driver, a packet is queued for transmission by the means of the `e1000_xmit_frame` function; 2) on QEMU, the `start_xmit` function is eventually triggered, which uses `process_tx_desc` auxiliary function to read the DMA-mapped transmission queue; 3) next, `xmit_seg`, `e1000_send_packet`, and `qemu_send_packet` functions are called in sequence, until 4) the packet is sent to be processed by the host's kernel.

Figure 17 illustrates these steps.

The packet reception process follows a different path: 1) when a packet destined to the guest arrives, the host OS forwards the packet to QEMU, eventually triggering the `e1000_receive` function; 2) next the `e1000_receive_iov` auxiliary function is used to queue incoming packets in the specific data structures; 3) the guest kernel polls the device for packets by the means of `e1000_clean` function; 4) `e1000_clean_rx_irq` translates packets into internal data structures; 5) finally, functions `e1000_receive_skb` and `napi_gro_receive` send the packets to the guest's kernel networking stack.

Figure 18 illustrates this process.

In terms of the adaptation to the SPDM protocol, the driver becomes the SPDM requester, whereas the E1000 virtual card becomes the responder[11]. Both counterparts have to bootstrap the SPDM protocol before heading into the application phase, in which packets are encrypted between the two endpoints.

Regarding the E1000 network card, internal SPDM variables are loaded during the virtual machine initialization (in the `pci_e1000_realize` function). After this point, the virtual device is ready to start processing SPDM messages.

[11] The modifications are incompatible two optimizations, namely TCP Segmentation Offload and Scatter-Gather, which had to be disabled.

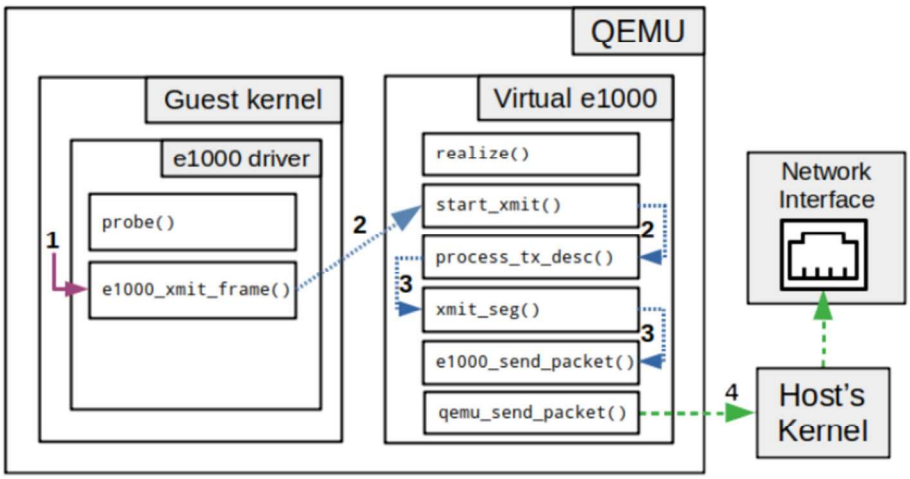

Fig. 17 Sending packets on an emulated E1000 network card

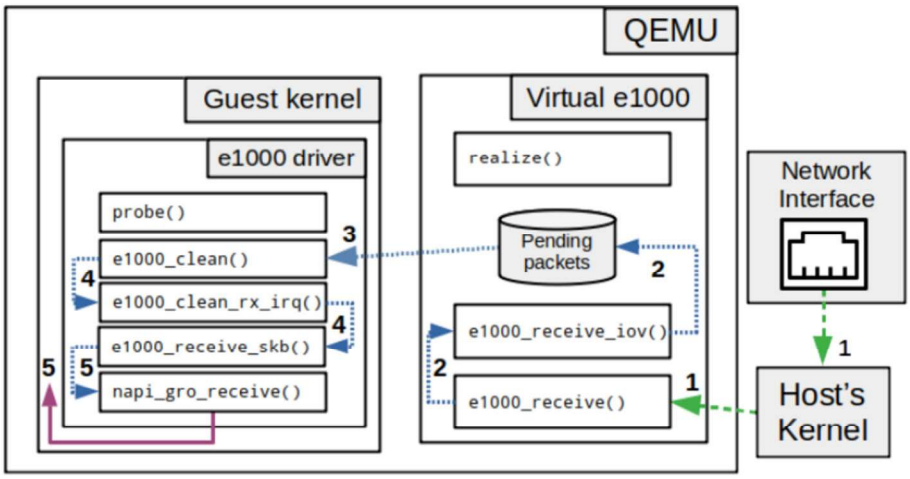

Fig. 18 Receiving packets on an emulated E1000 network card

At the guest OS, the kernel driver initializes basic SPDM-related data during the driver initialization (`e1000_open`), while the SPDM bootstrap procedure starts when a new interface is created (`e1000_probe`). SPDM messages are encoded into modified socket buffer structures similarly to regular packets, but using a special marker. By the end of `e1000_probe` function, a symmetric key is established, which can be used for encrypting application data.

Packets from the guest OS to the network are encrypted at the `e1000_xmit_frame` function, to be decrypted in QEMU by the virtual device at the `process_tx_desc` function. Similarly, packets from the network to the virtual machine are encrypted at the `e1000_receive_iov` function (on QEMU), and decrypted by the kernel driver within the `e1000_clean_rx_irq` function. Like the hard drive experiments, both kernel and QEMU are linked with `libspdm` library to provide SPDM functionality.

Appendix D Hard drive benchmark tools

We used widely employed tools and benchmarking utilities to assess hard drive performance: `dd`, `hdparm`, `ioping`, `bonnie++`, and `fio`.

Some of these tools provide built-in statistical data. The others were run multiple times and had their outcomes summarized manually by the research team. The usage, metrics provided, and output processing for the five tools employed are described in what follows.

The following QEMU command line parameters were used:

- `-enable-kvm`: enables KVM (Kernel-based Virtual Machine), which enhances virtual machine performance;
- `-cpu qemu64`: for selecting the emulated CPU model;
- `-kernel bzImage`: selects the kernel booted by the virtual machine. We used Linux Kernel version 4.19 from Buildroot;
- `-drive file=rootfs.ext2,if=ide,format=raw`: indicates the root file system used in the VM, using IDE

interface, and raw format. We used the root file system from Buildroot;
- `-append "console=ttyS0 rootwait root=/dev/sda"`: kernel command-line options. Sets the default console output, waits until root device is ready, and sets the root file system partition;
- `-m 1024`: virtual machine RAM, in megabytes;
- `-drive file=benchmarkdisk,if=virtio,format=raw`: appends a virtio-based additional hard drive, which is the target of our experiments.

D.1 dd (from BusyBox v1.31.1)

This is a commonplace utility found on Unix-like operating systems. Its primary use is to transfer raw data from one destination to another.

In our experiments, we used `dd` to test write speed. Specifically, we read data from `/dev/zero`, which is a fast source of dummy data, and wrote it to a file on the target disk.

We tested writing 2 gigabytes of data to the disk according to two approaches: in small blocks of 4KB each, or in 512MB-long blocks. In all cases, we enforced that the data was physically written on the device before the commands returned with the `conv=fsync` command line option. The write speed is calculated by the quotient between the total amount of data written and the time it takes to complete the operation. We used the `time` command to measure the execution time with `dd`. For each block size, the results hereby presented correspond to the average of 10 repetitions of the writing procedure.

D.2 hdparm (from BusyBox v1.31.1)

This command line tool is also commonly found in Linux systems. Besides using it to set and read hard drive parameters, we also explored its `-t` option switch, which provides buffered reading speed estimates. The tests with this tool were run a total of 10 times.

D.3 ioping v0.9

This is a tool for monitoring disk latency. It works similarly to the well-known tool from the network domain `ping`, i.e., by sending short requests to the disk and measuring how long they take to be fulfilled. We used the default parameters while testing both reading and writing latency, executing a total of 10 pings for each operation.

D.4 bonnie++ v1.04

This is a purpose-built benchmarking toolkit for hard drives. It automatically performs write, rewrite, and read tests. The metrics extracted from this tool were read and write speed measured in kB/s. The main command line options employed were:

- `-x 10` runs the benchmark 10 times;
- `-s 2G` specifies total amount of read/write data to 2 gigabytes;
- `-n 0` disables the file creation test, which is of little interest to our scenario because this test relates mostly to the file system;
- `-f` skips per-character tests, since our goal was to test HD behavior similar to common system usage;
- `-b` specifies unbuffered writes,
- `-D` uses the `O_DIRECT` flag, which attempts to perform requests synchronously.

D.5 fio v3.23

This is a highly customizable benchmarking tool for hard drives. Its main goal is to enable the creation of a workload as close as possible to the desired test case. Among the large set of metrics provided by this tool, we focused on I/O operations per second (iops). The main command line options used were:

- `--size=<size>` sets the portion of the disk that will be used to perform the tests. We used 2 GB in all experiments,
- `--io_size=<size>` total amount of data used in each I/O transaction. We used 5 GB in all experiments,
- `--rw=<option>` the type of I/O pattern. Common values are `read` (sequential reads), `write` (sequential writes), `randread` (random reads), and `randrw` (random reads and writes mixed),
- `--blocksize=<size>` the size of each individual operation. We used 1024 KB for sequential operation patterns and 4 KB for random operation patterns,
- `--fsync=<n>` issues a synchronization command at every `<n>` writes. We configured `<n>` to 10,000 for sequential operation patterns and 1 for random operation patterns.

The aforementioned set of command line options yields four different tests performed with the `fio` tool: 1) sequential reads with 1024 KB blocks; 2) sequential writes with 1024 KB blocks; 3) randomized reads with 4 KB blocks; and 4) randomized mixed reads and writes with 4 KB blocks.

D.6 boot time

Contrary to the other metrics hereby evaluated, we did not use any specialized tool to measure system boot time. Instead, we modified the guest's initialization scripts to log the sys-

tem uptime as the last step of the initialization process. The system uptime was obtained from reading `/proc/uptime`, a file that counts the seconds elapsed from the moment the kernel takes control of the CPU, yielding a precision of hundredths of a second. We collected a total of 15 boot times.

Appendix E Network card benchmark tools

All tests were executed on a QEMU virtual machine equipped with a E1000 network card. The following parameters were added to the QEMU command line to add the network card: `-device e1000,netdev=net0 -netdev user,id=net0,hostfwd=tcp::5555-:80`.

We used two benchmarking utilities to assess networking performance: `iperf3`, and `netperf`. Additionally, we set up web servers within the host and within the guest to check how long it takes to retrieve files of different sizes. Some of the tools used provide statistical data. Those that do not provide statistical data were run multiple times before having their outcomes summarized manually by the research team.

The usage, metrics provided, and output processing for the tools employed are described in the following subsections.

E.1 iperf3

This tool measures network throughput by sending data from a client to a server or vice versa. It is also instrumented to collect other metrics, such as CPU usage, packet loss rate, and jitter, depending on the selected protocol and flow direction.

The main `iperf3` server (executed in the host machine) command line options are:

- `-s` starts a server
- `-i 0` disables periodic reports
- `-J` selects JSON output

While command line options used at the client (executed in the guest machine) are:

- `-c <ip>` starts a client connection to `iperf3` server running at target IP address
- `-i 0` disables periodic reports
- `-J` selects JSON output
- `-udp` uses UDP rather than TCP
- `-R` reverses the direction of a test, so that the server sends data to the client
- `-t <n>` test length in seconds

The combination of protocols (TCP/UDP) and flow direction (client to server and server to client) yields four tests. The default test length is 10 seconds, however we decided to increase it to 60 seconds to reduce the weight of transient behavior. Since `iperf3` does not provide statistical data, each test was executed 10 times to calculate confidence intervals.

The metrics of interest are throughput, round trip time (RTT), and CPU utilization (both at the server and at the client). This tool also provides jitter and percentage of lost packages, but only for UDP tests.

E.2 netperf

This is another networking benchmark tool. Its functionality is similar to `iperf3`, but it is a completely separate implementation, with different command line options.

On a default Ubuntu installation, the `netperf` server (netserve binary) is automatically started along with the OS initialization process. No additional configuration is needed.

The `netperf` client ran at the guest virtual machine. It has several command line options, we chose to work with the following subset:

- `-c -C` measures CPU usage at client and server
- `-I 95,5` calculates 95% confidence intervals, while specifying a 5% confidence interval width is desired
- `-j` keeps additional timing statistics
- `-H <ip>` connects to netserver at the target IP address
- `-t omni` sets test type to 'omni'
- `--` separator from global options to test specific options
- `-d <dir>` sets test direction (send, receive, or rr)
- `-T <proto>` sets protocol type. We will focus on UDP and TCP
- `-o "<c1>,<c2>,.."` sets output format to CSV, listing the output columns

Appendix F Numeric values

Table 2 summarizes all numeric values obtained in all performed experiments with the hard drive, while Table 3 summarizes all numeric values obtained in all experiments with the network card.

Table 2 Hard drive benchmarks numeric results

Measurement	With SPDM		Without SPDM	
	Average	Standard deviation	Average	Standard deviation
dd small blocks [B/s]	2.59E+04	145	1.51E+05	1.31E+04
dd big blocks [B/s]	2.71E+04	60	1.52E+05	1.03E+04
ioping read latency [us]	866	335	1.08E+03	2.49E+03
ioping write latency [us]	1.58E+04	7,677	1.84E+04	1.11E+04
hdparm read speed [kB/s]	2.07E+04	145	3.83E+06	1.03E+06
bonnie++ read speed [kB/s]	1.75E+04	50	2.24E+06	2.43E+04
bonnie++ write speed [kB/s]	1.28E+04	50	1.08E+05	6.57E+03
fio sequential read [iops]	19.9	1.0	5.94E+03	249
fio sequential write [iops]	17.5	0.9	1.99E+03	1.41E+03
fio random read [iops]	3.94E+03	354	2.16E+04	192
fio random rw read [iops]	22.5	8.7	22.6	8.7
fio random rw write [iops]	22.5	2.7	22.6	2.6

Table 3 Network card benchmark numeric results

Measurement	With SPDM		Without SPDM	
	Average	Standard deviation	Average	Standard deviation
iperf3: tcp receive throughput [Mbps]	89.17	0.13	509.4	2.1
iperf3: tcp receive rtt [us]	2,965	80	1,197	564
iperf3: tcp receive guest CPU utilization [%]	60.05	0.21	31.0	0.2
iperf3: tcp receive host CPU utilization [%]	0.161	0.003	0.64	0.01
iperf3: tcp send throughput [Mbps]	89.94	0.19	578.2	2.6
iperf3: tcp send rtt [us]	385	24	82.9	2.6
iperf3: tcp send guest CPU utilization [%]	99.66	0.05	38.5	2.2
iperf3: tcp send host CPU utilization [%]	3.9	0.4	15.8	4.9
iperf3: udp receive throughput [Mbps]	73.8	6.4	95.5	3.0
iperf3: udp receive jitter [ms]	3.4	1.3	1.9	1.2
iperf3: udp receive lost packets [%]	98.68	0.11	98.39	0.07
iperf3: udp receive guest CPU utilization [%]	10.06	0.88	3.93	0.14
iperf3: udp receive host CPU utilization [%]	98.72	0.03	98.72	0.04
iperf3: udp send throughput [Mbps]	89.68	0.22	513.1	5.8
iperf3: udp send jitter [ms]	0.0118	0.0080	0.0089	0.0013
iperf3: udp send lost packets [%]	0.0	0.0	0.0	0.0
iperf3: udp send guest CPU utilization [%]	99.69	0.04	99.66	0.03
iperf3: udp send host CPU utilization [%]	3.82	0.03	20.58	0.17
netperf: TCP Send throughput [Mbps]	86.83	0.74	616.0	1.1
netperf: TCP Send guest CPU utilization [%]	100.00	0.00	97.35	0.55
netperf: TCP Send host CPU utilization [%]	14.6	6.0	25.2	4.9
netperf: TCP Send latency [us]	1,509	277	209	306
netperf: TCP Receive throughput [Mbps]	88.5	1.1	510.41	0.28
netperf: TCP Receive guest CPU utilization [%]	79.7	4.8	44.1	5.4
netperf: TCP Receive host CPU utilization [%]	15.9	2.2	17.3	1.9
netperf: TCP Receive latency [us]	785	696	135	74
netperf: TCP SendlRecv throughput [Trans/s]	7,304.76	0.48	7,383.88	0.63

Table 3 continued

Measurement	With SPDM		Without SPDM	
	Average	Standard deviation	Average	Standard deviation
netperf: TCP Send|Recv guest CPU utilization [%]	69.0	4.8	50.1	2.3
netperf: TCP Send|Recv host CPU utilization [%]	18.6	2.9	16.2	3.4
netperf: TCP Send|Recv latency [us]	136	104	135	120
netperf: UDP Send throughput [Mbps]	105.3	1.5	1,312.4	2.5
netperf: UDP Send guest CPU utilization [%]	100.0	0.0	100.0	0.0
netperf: UDP Send host CPU utilization [%]	15.5	4.6	15.1	4.7
netperf: UDP Send latency [us]	4,970	745	395	193
netperf: UDP Send|Recv throughput [Trans/s]	7,296.57	0.45	7,349.50	0.50
netperf: UDP Send|Recv guest CPU utilization [%]	67.4	3.8	49.9	5.0
netperf: UDP Send|Recv host CPU utilization [%]	18.8	3.8	16.4	3.2
netperf: UDP Send|Recv latency [us]	136	70	135	107

Acknowledgements The authors thank Luca Beraldo Basílio, Luca Bevilacqua Previato Roja, and Marcos de Souza Boger for providing an early draft of the SPDM-enabled network card code.

Author Contributions R.C.A.A. executed the experiments, generated the graphs, and wrote the main manuscript text. M.A.S.Jr. and B.C.A. idealized the paper and participated in experiment design. All authors reviewed the manuscript.

Funding The authors would like to thank Hewlett Packard Enterprise for supporting this project. This project was partially funded by Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP grant 2020/09850-0), Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq grant 304643/2020-3), and Fundação Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES Finance Code 001).

Availability of data and materials Any data can be made available upon request. Code used in the research is freely available on github (https://github.com/rcaalves/spdm-benchmark).

Declarations

Conflict of interests The authors have no relevant financial or non-financial interests to disclose.

Ethical approval Not applicable.

References

1. Cui, A., Costello, M., Stolfo, S.J.: When firmware modifications attack: a case study of embedded exploitation. In: Network and Distributed System Security Symposium (NDSS'13), p. 13. The Internet Society, San Diego, California (2013)
2. Brown, D., Walker, T., III., Blanco, J., Ives, R., Ngo, H., Shey, J., Rakvic, R.: Detecting firmware modification on solid state drives via current draw analysis. Comput. Secur. **102**, 102149 (2021)
3. Choi, B.-C., Lee, S.-H., Na, J.-C., Lee, J.-H.: Secure firmware validation and update for consumer devices in home networking. IEEE Trans. Consum. Electron. **62**(1), 39–44 (2016). https://doi.org/10.1109/TCE.2016.7448561
4. Menn, J.: NSA Can Hide Spyware in Hard-Disk Firmware. https://www.vox.com/2015/2/17/11559082/nsa-can-hide-spyware-in-hard-disk-firmware. Accessed 16 Dec 2021 (2015)
5. DMTF: DSP0274: Security protocol and data model (SPDM) specification, v.1.3.0. Technical report, Distributed Management Task Force (Jun 2023). www.dmtf.org/sites/default/files/standards/documents/DSP0274_1.3.0.pdf
6. Armellin, A., Caviglia, R., Gaggero, G., Marchese, M.: A framework for the deployment of cybersecurity monitoring tools in the industrial environment. IT Professional **26**(4), 62–70 (2024). https://doi.org/10.1109/MITP.2024.3396356
7. Cremers, C., Dax, A., Naska, A.: Formal analysis of SPDM: Security protocol and data model version 1.2. In: 32nd USENIX Security Symposium (USENIX Security 23), pp. 6611–6628 (2023)
8. Profentzas, C., Günes, M., Nikolakopoulos, Y., Landsiedel, O., Almgren, M.: Performance of secure boot in embedded systems. In: 15th DCOSS, pp. 198–204 (2019). https://doi.org/10.1109/DCOSS.2019.00054
9. Khalid, O., Rolfes, C., Ibing, A.: On implementing trusted boot for embedded systems. In: 2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), pp. 75–80 (2013). https://doi.org/10.1109/HST.2013.6581569
10. Yin, H., Dai, H., Jia, Z.: Verification-based multi-backup firmware architecture, an assurance of trusted boot process for the embedded systems. In: 2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications, pp. 1188–1195 (2011). https://doi.org/10.1109/TrustCom.2011.160
11. Kumar, V.B.Y., Gupta, N., Chattopadhyay, A., Kasper, M., Krauß, C., Niederhagen, R.: Post-quantum secure boot. In: 2020 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 1582–1585 (2020). https://doi.org/10.23919/DATE48585.2020.9116252
12. Dasari, S., Madipadga, V.: Aegis: A framework to detect compromised components in the supply chain of information technology infrastructure. In: IWBIS, pp. 159–164 (2020). https://doi.org/10.1109/IWBIS50925.2020.9255660
13. Velozo, F., Ferreira, T., Pacheco, E., Alves, R., Jr., M.S., Albertini, B., Batista, D.: Fuzzing para o protocolo TLS: Estado da arte e comparação de fuzzers existentes. In: XXIII SBSeg, pp. 303–308. SBC, Porto Alegre, RS, Brasil (2023). https://doi.org/10.5753/sbseg_estendido.2023.235133
14. Ferreira, T.D., Freitas, O.F., Alves, R.C.A., Albertini, B.C., Jr., M.A.S., Batista, D.M.: SPDM-WiD: Uma ferramenta para

inspeção de pacotes do security protocol data model (SPDM). In: XLII SBRC, p. 8. SBC, Porto Alegre, RS, Brasil (2024)
15. Fujimoto, A., Peterson, P., Reiher, P.: Comparing the power of full disk encryption alternatives. In: 2012 International Green Computing Conference (IGCC), pp. 1–6 (2012). IEEE
16. Brož, M., Patočka, M., Matyáš, V.: Practical cryptographic data integrity protection with full disk encryption. In: IFIP International Conference on ICT Systems Security and Privacy Protection, pp. 79–93 (2018). Springer
17. Petschick, M.: Full disk encryption on unmanaged flash devices. Master's thesis, Technische Universität Berlin, Germany (2011)
18. Bosen, B.: Full drive encryption with Samsung solid state drives. Technical report, Trusted Strategies LLC (2010)
19. Bosen, B.: FDE performance comparison - hardware versus software full drive encryption. Technical report, Trusted Strategies LLC (2010)
20. Mayrhofer, R.: An architecture for secure mobile devices. Secur. Commun. Netw. **8**(10), 1958–1970 (2015). https://doi.org/10.1002/sec.1028
21. Alves, R.C.A., Albertini, B.C., Simplicio, M.A.: Securing hard drives with the security protocol and data model (spdm). In: 2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pp. 446–447 (2022). https://doi.org/10.1109/ISVLSI54635.2022.00099
22. Alves, R.C.A., Jr., M.A.S., Albertini, B.C.: Adapting a network card and a hard drive to SPDM. GLOBECOM 2022 - 2022 IEEE Global Communications Conference (2022)
23. QEMU: QEMU - A generic and open source machine emulator and virtualizer. https://www.qemu.org/
24. Kukunas, J.: Chapter 8 - perf. In: Power and Performance, pp. 137–165. Morgan Kaufmann, Boston (2015). https://doi.org/10.1016/B978-0-12-800726-6.00008-2
25. DMTF: libspdm is a sample implementation that follows the DMTF SPDM specification. https://github.com/DMTF/libspdm/