

# Benchmarking the Security Protocol and Data Model (SPDM) for component authentication

Renan C. A. Alves, Bruno C. Albertini, Marcos A. Simplicio Jr

**Abstract**—Efforts to secure computing systems via software traditionally focus on the operating system and application levels. In contrast, the Security Protocol and Data Model (SPDM) tackles firmware level security challenges, which are much harder (if at all possible) to detect with regular protection software. SPDM includes key features like enabling peripheral authentication, authenticated hardware measurements retrieval, and secure session establishment. **Since SPDM is a relatively recent proposal, there is a lack of studies evaluating its performance impact on real-world applications.** In this article, we address this gap 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 based on VirtIO, and comparing the resulting read/write performance with a regular, unsecured implementation. Our results suggest that SPDM bootstrap time takes the order of tens of milliseconds, while the toll of introducing SPDM on hard drive communication highly depends 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.

**Index Terms**—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 in an assembly line. After assembly, devices are tested and then shipped to warehouses, eventually reaching the end user's 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 substitute components** [1], [2]. The goals of such attacks may vary. For example, they may consist in eavesdropping sensitive data stored in volatile or non-volatile memory, as well as passing through video, audio or network components. They may also involve modifying the default behavior of components to to enforce built-in obsolescence or impose some kind of censorship.

Although it may seem that such attacks at the hardware/firmware level belong to the realm of fiction, the existence of successful proof-of-concept implementations indicate otherwise. For example, a vulnerability in the remote firmware update functionality of a family of printers allowed arbitrary code to be injected into the printer firmware [1]. The authors then developed a self-replicating malware capable of eavesdropping on data and performing network reconnaissance. Another attack, this time targeted at USB firmware, enabled a device (e.g., a thumb drive) to impersonate a different device type (e.g. a keyboard) [3]. Even more concerning is the fact that evidence of firmware tampering has also been found outside the halls of academia. In 2015, for example, hard drives from various

manufacturers had their firmware altered in such a manner that the modified code could be used to retrieve data even from encrypted partitions [4].

Protecting against such hardware-level tampering attacks is a challenging task. In particular, 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 recently proposed by the DMTF (Distributed Management Task Force) to address such low-level security challenges. 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 been victim of tampering, while establishing sessions keys avoids passive eavesdropping by malicious components attempting to steal data.

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 the end-users' experience and identifying bottlenecks that might be optimized in the protocol. Specifically, we quantify the overhead added by the security layer to size the impact on end-user experience, and to identify protocol bottlenecks that might be optimized, using an emulated environment. The resulting contributions are twofold: (1) we evaluate the overhead of each individual phase of the SPDM protocol execution on an extremely simple SPDM-enabled component, a random number generator (Section 3); and (2) we build an SPDM-enabled hard drive and assess the performance impact on userspace-perceived reading and writing speeds

• The authors are with Universidade de São Paulo, Brazil.

(Section 4). 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 *66ms*. 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.

The rest of this manuscript is organized as follows. Section 2 summarizes SPDm’s key aspects. We study the individual overhead of each SPDm message in Section 3, where we describe our methods and results. Section 4 contains specification and results regarding our SPDm-enabled hard drive. Finally, we discuss related work in Section 5, and present closing remarks in Section 6.

## 2 SPDm

This section summarizes basic concepts and workflow of SPDm in its version 1.1 [5].

The SPDm is a proposed standard for secure inter-communication 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 encapsulation approach employed for conveying those messages.

The SPDm standard defines three pairs of mandatory messages, 14 pairs of facultative messages, and additional messages to handle errors. A set of those messages are related to the core SPDm 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 cryptographic algorithms to be employed, and maintaining an active communication session. Figure 1 gives an overview of the expected message flow, highlighting which messages are mandatory. Since **each individual message is evaluated in our experiments**, we briefly describe them in what follows.

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 requester and 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 operation.

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 has 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/CHALLENGE\_AUTH message pair. In this process, the responder may also indicate in its CHALLENGE\_AUTH response that it would like to perform a mutual authentication, as long as this feature is supported by both responder and requester. Mutual authentication follows the same procedure employed when authenticating the responder toward the requester (i.e., with the messages for fetching digests, requesting certificate, and issuing a challenge), but reversing the roles of each communicating component. Since the responder typically does not send asynchronous messages to the requester, though, encapsulated messages are used instead: the requester sends a GET\_ENCAPSULATED\_REQUEST message to start the communication; the responder then answers with a ENCAPSULATED\_REQUEST message, which triggers a DELIVER\_ENCAPSULATED\_RESPONSE reply by the requester; finally, the responder acknowledges the reception of this latter message by means of a ENCAPSULATED\_RESPONSE\_ACK message, which may itself contain another encapsulated request if necessary.

A measurement may represent firmware, software, or some 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. These messages rely on the responder being provisioned with at least one certificate chain considered valid by the requester. 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. After a session is established, the underlying keys can be updated with the KEY\_UPDATE and KEY\_UPDATE\_ACK messages; both requester and responder may initiate this key update process. 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. When the session must be terminated, though, this can be accomplished by means of the

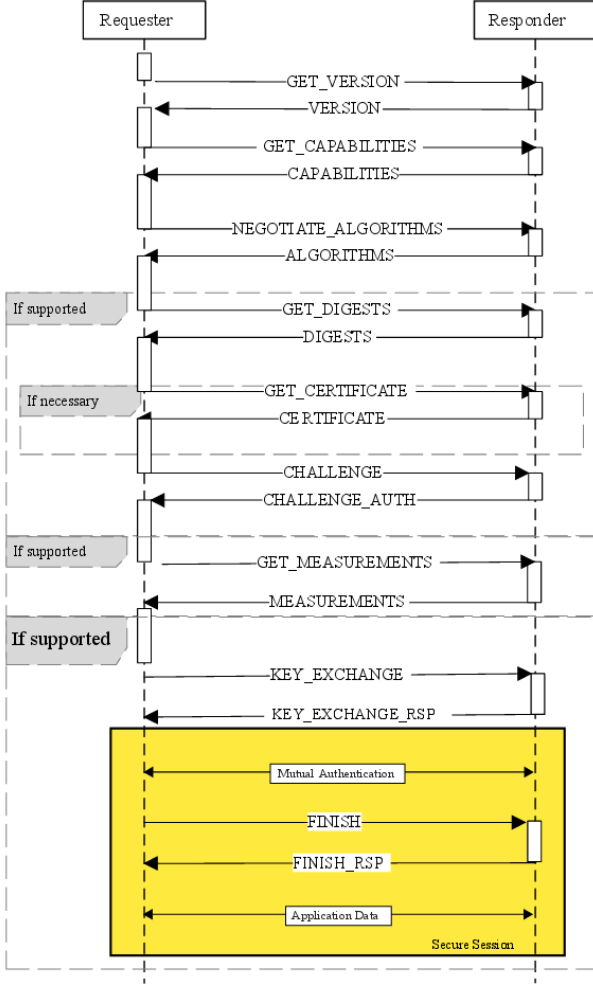


Fig. 1. SPDM message flow. Extracted from [5].

END\_SESSION/END\_SESSION\_ACK messages.

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

### 3 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 supports two operation modes: 1) secure mode, which follows the SPDM specification, and 2) clear-text mode, without any SPDM-related security capabilities.

#### 3.1 Method

We used a virtualized experimental setup based on the QEMU emulation software [6]. 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. After initializing, the RNG waits until a device driver writes a request message to the MMIO region, and then indicates

that the message is ready by writing to a control register. After reading the message, the RNG device sends an interrupt signal to announce that the response can be read from the MMIO region.

Non-SPDM transactions were used as a baseline for estimating the overhead introduced by SPDM to this simple procedure. These transactions occur over specific memory region addresses, distinct from the region used for SPDM transactions.

We implemented an SPDM requester as a device driver for our RNG on a Linux-based operating system. This device driver is built upon the userspace I/O system (UIO). Figure 2 depicts a schematic of our setup. We note that, at least in principle, SPDM should be oblivious to the operating system. Nonetheless, the setup described provides a controlled and malleable environment, suitable for yielding meaningful performance results from our prototype implementation.

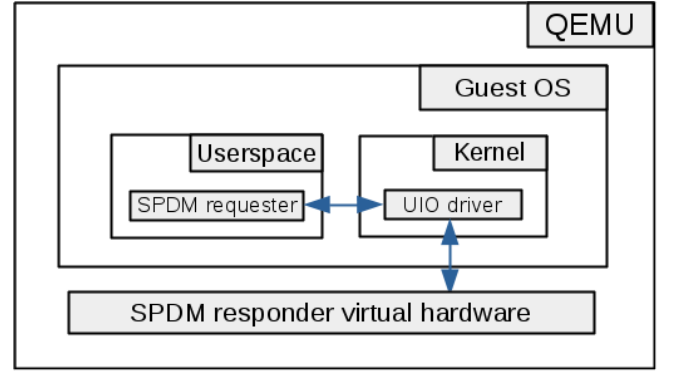


Fig. 2. System architecture

In our experiments, the interaction between the RNG device and its device driver was conducted following five steps: 1) SPDM local initialization (memory allocation, library configuration); 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: retrieving each measurement individually and retrieving all of them at once. In the first case, 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. Either way, the responder was configured to hold 5 different measurements, following the DMTF measurement block specification (SPDM specification v1.1.0 [5], Section 10.11.1.1). Each block contains 128 bytes of dummy data.

The application phase is also divided in 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 [7], allowing us to as-

sess 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 and it can be deceiving in an emulated environment. 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). It is also worth mentioning that the `perf` API does not give perfect measurements, since the results provided include some overhead introduced by itself; this influence is, however, observed mostly on short measurements.

SPDM capability was provided by the `libspdm` open source library [8]. For better reproducibility, `libspdm` compile-time and execution-time parameters used in the experiment are listed in Table 1.

TABLE 1  
libspdm parameters

Parameter	Value
SPDM protocol version	1.1
libspdm version	commit dc48779
libspdm build options	x64, release
Underlying crypto library	MbedTLS
Requester signature algorithm	TPM_ALG_RSAPSS_3072
Responder signature algorithm	TPM_ECC_NIST_P384
Measurement hash algorithm	TPM_ALG_SHA384
DHE algorithm	SECP_384_R1
AEAD algorithm	AES_256_GCM
Key scheduling algorithm	As defined by SDPM v1.1
Mutual authentication	enabled

As emulation environment, we used QEMU version 4.1 [6]. A QEMU virtual machine is specified by command-line options: hard drives, CPU, network cards, along with other options. The chosen guest operating system is based on `qemu_x86_64_defconfig` preset configuration from Buildroot<sup>1</sup> version 2020.02.9, which contains Linux Kernel version 4.19. Also for better reproducibility, the exact 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 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,

1. Available at <https://buildroot.org/>

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

The host system ran a Linux-based system on an Intel i7-10700KF processor, with 3800MHz clock, 8 Cores, 16 logical processors, and 32 GB of RAM. Ideally, the CPU clock should be constant while performing benchmarks. Hence, aiming to approach this ideal scenario, the following options were disabled at the machine BIOS configuration menu: turbo boost, speed step, speed shift, hyper threading, and CPU C states.

Each of execution step was performed 100 times, aiming to obtain statistical confidence. The graphs shown in Sections 3.2 and 3.3 present the average value of all runs along with 95% confidence intervals.

### 3.2 Results: requester

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

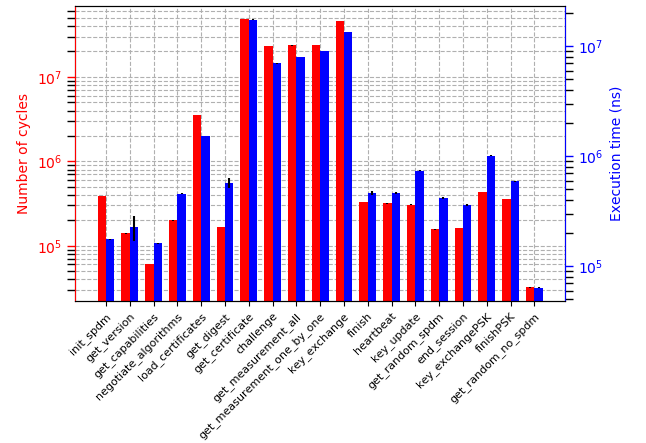


Fig. 3. Requester execution time and number of cycles.

Figure 3 shows that the most time consuming messages are `GetCertificate` and `KeyExchange`, taking respectively 17.4 and 13.5ms, or 47.9 and 45.9 million cycles on average. The `GetCertificate` procedure is expected to be slow since (1) it may require several messages to finish, and (2) by the end of it, the retrieved certificate must be verified for correctness, which requires a few signature verifications. `KeyExchange`, in turn, involves the generation of a symmetric key pair by means of a Diffie-Hellman key exchange. Also, our analysis considers that the communication parties engage in mutual authentication, which can be considered a worst case scenario in terms of performance.

The next most time consuming messages are `GetMeasurements` and `Challenge`, both of which take around 23.5 million of cycles to execute. The reason



is that, in both cases, this overhead refers essentially to their underlying signature verification procedure, so it is reasonable for them to take roughly the same amount of time. Interestingly, though, we noticed that retrieving measurements all at once is slightly faster than retrieving measurements one by one. Specifically, the time taken in the former case is  $7.9ms$ , against  $9.0ms$  in the latter. This represents a time gain of 11.9%, while the number of cycles is reduced by 2.8%.

The usage of Pre Shared Keys (PSK), on the other hand, considerably reduces the burden of establishing session keys. The precise difference is that `KeyExchangePSK` takes only  $1.0ms$  ( $4.3 \times 10^5$  cycles), which is only a fraction of the  $13.5ms$  ( $4.6 \times 10^7$  cycles) observed in its asymmetric counterpart. Also, using a PSK setting allows components to forego `GetCertificate` and `Challenge` messages, further speeding up the process.

All other messages take from 161 up to  $738us$  on average, depending on their underlying complexity, so they are unlikely to become bottlenecks in practice. Conversely, we notice that the task of loading the certificate from the disk can take a significant amount of time:  $1.5ms$ , or  $3.6 \times 10^6$  cycles, on average.

Once a secure session is established, we were able to retrieve a random number from the SPDM-enabled RNG device in  $415us$  ( $1.6 \times 10^5$  cycles). Conversely, in our SPDM-free baseline execution the same operation took  $63us$  ( $3.2 \times 10^4$  cycles) on average. This means that SPDM led to a 6.4-fold increased in terms of time, or a 4.8-fold increase in number of cycles. This result is not surprising, though, when we take into account the remarkable simplicity of the device evaluated in our tests. After all, our simple RNG was designed to be extremely fast, so even cryptographic operations that are quite lightweight in absolute numbers, like symmetric encryption, become comparatively expensive.

### 3.3 Results: responder

Figure 4 shows metrics extracted from the Responder. As shown in this figure, `KeyExchange` is the most expensive message to process, taking  $10.1ms$ , or  $3.8 \times 10^7$  cycles. As expected, though, adopting a PSK setting reduces the `KeyExchange` overhead to only  $52us$  or  $1.8 \times 10^5$  cycles.

In comparison with the requester, the responder handles `GetCertificate` messages faster than `KeyExchange`. The reason behind this behavior is that most of the cryptographic processing of `GetCertificate` remains at the requester side. At the responder, `GetCertificate` takes  $17us$ , or  $4.2 \times 10^4$  cycles.

Similarly to the requester, `GetMeasurements` and `Challenge` are nearly tied as the second most time consuming operations at the responder, taking approximately 13 million cycles or  $3.5ms$ . Once again, retrieving measurements all at once was slightly faster than retrieving one by one. More precisely, those operations take respectively  $3.5$  and  $3.6ms$ , which means a 1.5% gain in terms of time costs (or a 1.1% reduction in the number of cycles).

The largest overhead observed at the responder, however, was the time to load certificates from the disk, which took  $13.8ms$  or  $5.2 \times 10^7$  cycles. The discrepancy between responder and requester is caused by essentially by our

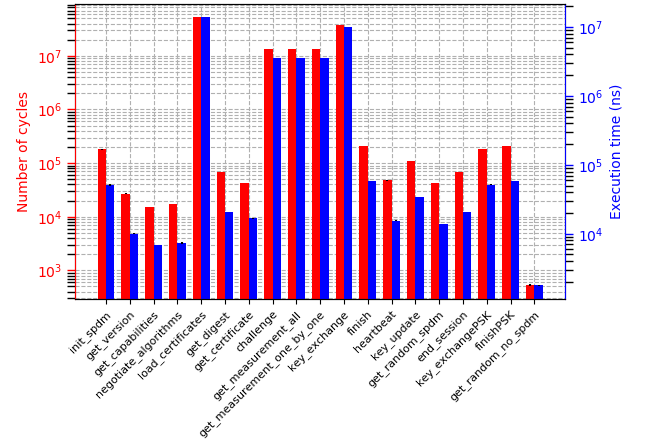


Fig. 4. Responder execution time and number of cycles.

configuration, where each party uses a different signature algorithm. More precisely, the responder uses a signature algorithm based on elliptic curves, which takes longer to verify than the RSA-based signatures generated by the requester.

Processing a random number request without SPDM took the responder  $1.8us$ , or 533 cycles. Adding the SPDM layer increases this cost to  $13.8us$ , or  $4.2 \times 10^4$  cycles. The processing of all other messages took from  $6.8us$  to  $58us$  (or  $1.5 \times 10^4$  to  $2.1 \times 10^5$  cycles), adding a moderate amount of overhead to the protocol.

## 4 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 its importance and to the availability of tools for conducting such performance tests. Specifically, we compared an SPDM-equipped hard drive to a regular, unsecured one, considering as metrics: boot time, read speeds, and write speed, under various workloads.

### 4.1 Implementation details

Due to the lack of off-the-shelf SPDM-enabled hardware, we once again resorted to emulation. One more time, we used QEMU as 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_blk` hard drive. The guest operating system, in turn, is a custom Buildroot-based Linux distribution. Its kernel contains a native `virtio_blk` driver, compatible with QEMU's `virtio_blk` hard disk. Both driver and hard disk were then modified to incorporate SPDM security functionalities, as provided by `libspdm`.

In short, the interaction between kernel driver and virtual device is as follows: 1) the operating systems sends read/write requests<sup>2</sup> to the driver queue (`queue_request`

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

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 Figure 5 illustrates these steps.

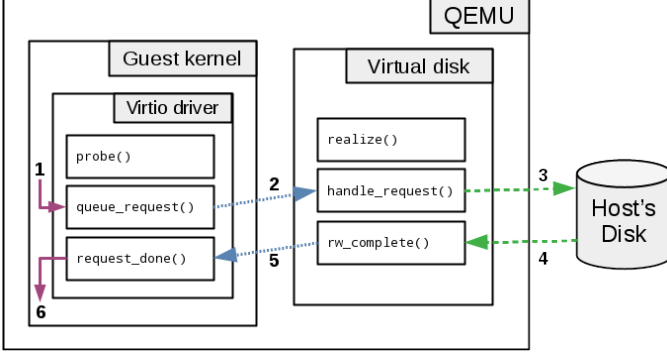


Fig. 5. `virtio_blk` driver and virtual hardware interaction: 1) Guest operating system request, 2) Driver request to virtual hardware, 3) Request forwarded to host, 4) Request result at virtual hardware, 5) Request result at driver, 6) Request finished

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 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 takes place. As a result, all SPDM messages, from `GET_VERSION` to `KEY_EXCHANGE`, are exchanged at this moment. 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.

Within both kernel and QEMU, SPDM functionalities are provided by `libspdm` using the same parameters employed in Section 3.1, as summarized in Table 1. Minor adjustments were needed to deal with stack overflow issues when calling

some `libspdm` functions, though, caused by the limited stack space available within the Linux kernel.

## 4.2 Method

In our experiments, we used a few widely employed tools and benchmarking utilities to assess hard drive performance: `dd`, `hdparm`, `ioping`, `bonnie++`, `fio`, which are further discussed separately at the end of this section. All tests were executed on a QEMU virtual machine, on a separately attached hard drive using the `virtio_blk` interface. 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. In both scenarios, 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.

All experiments were conducted to reach statistical significance. Some of the tools used provide statistical data, while others that do not were run multiple times and had their outcomes summarized manually by the research team to achieve this goal. The usage, metrics provided, and output processing for the five tools employed are described in what follows.

### 4.2.1 `dd` (from *BusyBox v1.31.1*)

this is a commonplace utility found on Unix-like operating systems. Its primary usage 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 for 10 repetitions of the writing procedure.

#### 4.2.2 `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 explore its `-t` option switch, which provides buffered reading speed estimates. The tests with this tool were run a total of 10 times.

#### 4.2.3 `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.

#### 4.2.4 `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 file creation test, which is of little interest to our scenario because since this relates mostly to the file system;
- `-f` skips per-character tests, since our goal was to test HD behavior that is close to common system usage;
- `-b` specifies unbuffered writes,
- `-D` uses the `O_DIRECT` flag, which attempts to perform requests synchronously.

#### 4.2.5 `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>` as 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) random reads with 4 KB blocks; and 4) mixed reads and writes with 4 KB blocks.

#### 4.2.6 `boot time`

contrary to the other metrics hereby evaluated, we did not use any specialized tool to measure the system boot time. Instead, we modified the guest's initialization scripts to log the system uptime as the last step of the initializing 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.

### 4.3 Experimental Results

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

#### 4.3.1 `dd`

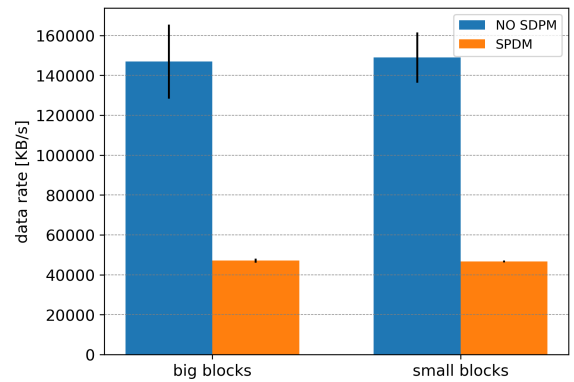


Fig. 6. Measuring data rate of an SPDM-enabled hard disk: `dd`

Figure 6 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 on SPDM's impact over disk writing speed when such workloads are prevalent, it also serves as baseline for scenarios where random disk accesses are more common.**

In general, writing small blocks tends to be slightly slower than writing large blocks, which was observed in our results ( $\approx 1\%$  slower). **In both cases, though, SPDM caused a  $\approx 68\%$  slowdown in writing speed.**

#### 4.3.2 `hdparm`

Quoting its manual, the `hdparm` test provides “an indication of how fast the drive can sustain sequential data reads under Linux, without any filesystem overhead.”. Without SPDM, the average read speed observed was  $3.9\text{GB/s}$ , fairly close to the nominal  $6\text{GB/s}$  speed of the hard drive, considering the virtualization overhead. Introducing SPDM, though, drastically decreases the value indicated by `hdparm` to  $28\text{kB/s}$ , which translates to a 99.3% speed degradation.

#### 4.3.3 ioping

Figure 7 shows read and write latency results according to *ioping*. Focusing on average results only, the introduction of SPDM increased the reading latency by 208%, while it decreased writing latency by 39%. However, the obtained confidence intervals in both cases were very large, making it hard to draw statistically relevant conclusions with this tool.

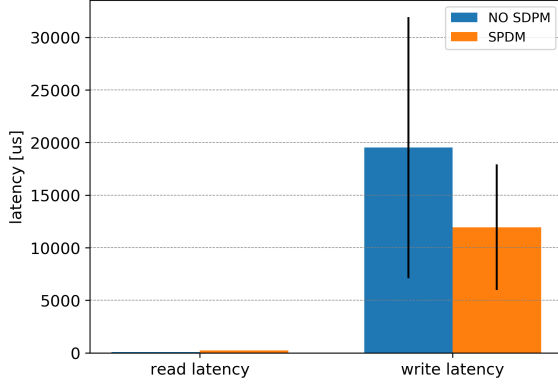


Fig. 7. Measuring latency of an SPDM-enabled hard disk: *ioping*

#### 4.3.4 bonnie++

The results from *bonnie++* (Figure 8) indicate a harsh loss of performance when SPDM is introduced. The tests performed by this tool consist of reading and writing 200Mb files to the disk. The writing portion is somewhat similar to the one performed 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 115MB/s to 23MB/s (a 79.8% reduction). The loss of reading performance is even more significant: from 2.0GB/s to only 28MB/s. It makes sense that both reading and writing speed drop to the same order of magnitude, since the system bottleneck is the same in both tests – the processing cost of encrypting/decrypting every transaction.

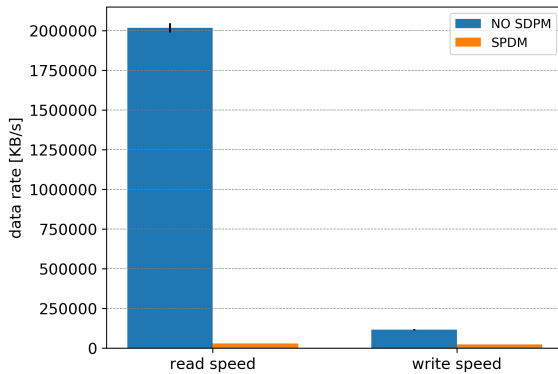


Fig. 8. Measuring data rate of an SPDM-enabled hard disk: *Bonnie++*

#### 4.3.5 fio

Results from the *fio* tool are presented in Figure 9, 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 synchronization sparsely. In this case, the pattern observed is similar to the one 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.

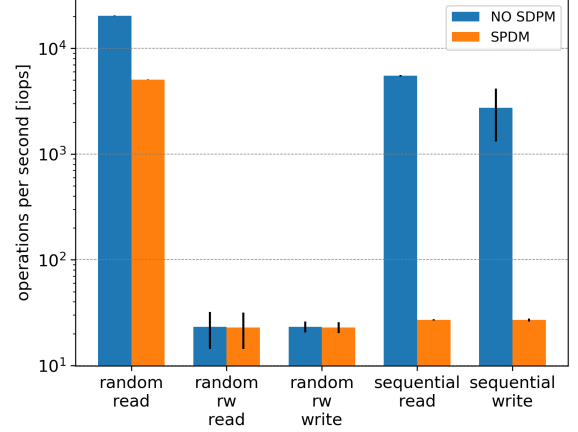


Fig. 9. Measuring transaction rate of an SPDM-enabled hard disk: *fio*

The performance degradation becomes less prominent when randomness is introduced. For these tests, the block size was 4kB, and requests to synchronize data were sent after every operation. When performing random read-only tests, the performance drops to 25.0% of the baseline value. Mixing random reading and random writing (in Figure 9, label “random rw read” refers to read speed, and label “random rw write”, refers to read speed) yields virtually the same level of iops: introducing SPDM causes a reduction of approximately 1.3% 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 disk operations, namely the frequent seek operations to address the random request locations.

#### 4.3.6 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 66ms (3.48s to 3.55s), which resonates with the results presented in Section 3.

#### 4.3.7 Summary

Table 2 summarizes all numeric values discussed in this section.

## 5 RELATED WORK

SPDM is a fairly new standard, so its impact and benefits 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 pre-OS stages. In particular, SPDM is somewhat close



TABLE 2  
Hard drive benchmarks numeric results

	Average W/ SPDM	Standard Deviation	Average w/o SPDM	Standard Deviation
dd small blocks [kB/s]	4.66E+04	466	1.49E+05	1.27E+04
dd big blocks [kB/s]	4.71E+04	1.01E+03	1.47E+05	1.86E+04
ioping read latency [us]	234	12.0	76.0	4.00
ioping write latency [us]	1.19E+04	5.98E+03	1.95E+04	1.24E+04
hdparm read speed [kB/s]	2.81E+04	126	3.93E+06	7.29E+05
bonnie read speed [kB/s]	2.83E+04	131	2.02E+06	2.89E+04
bonnie write speed [kB/s]	2.32E+04	30.1	1.15E+05	4.22E+03
fio sequential read [iops]	27.0	0.315	5.52E+03	86.2
fio sequential write [iops]	26.9	0.892	2.74E+03	1.43E+03
fio random read [iops]	5.06E+03	15.6	2.02E+04	1.60E+02
fio random rw read [iops]	22.9	8.63	23.2	8.90
fio random rw write [iops]	23.0	2.70	23.3	2.75
boot time [s]	3.55	1.04	3.48	1.04

to techniques often called “secure boot” or “trusted boot”, whose goal is to ensure that only legitimate initialization firmware and bootloaders are executed.

The literature includes a number of studies on secure boot performance. For example, Profentzas et. al [9] 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 technique and algorithms employed.

In a similar fashion, Khalid et. al [10] proposed a trusted boot architecture for embedded systems based on an independent security processor. Their design was implemented on FPGAs, and their experiments showed that their secured boot process increases boot time by 25%, considering a Linux system customized for their needs.

The study by Yin et. al [11] brought attention to failure-prone NAND flash chips, commonly used to store bootloaders in embedded platform. The authors propose a redundancy scheme that verifies the integrity of bootloader firmware code and falls back to an alternative copy in case of checksum mismatch. Their experiments show that total boot time is increased by 65% if the bootloader is intact. Otherwise, it is increased by 255%, since the backup code has to be copied and verified once again.

Kumar et. al [12] 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, Dasari and Madipagda [13] stands out for being one of the few works in the literature that 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 individual components. However, their solution is based on a older version of SPDM, which did not yet support session key establishment. Consequently, the solution does not prevent passive sniffers from eavesdropping 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.

When considering the particular scenario of protecting communications from/to hard disks, this work shares a relationship with studies covering disk encryption technologies. Examples include works that focus on the energy consumption of full disk encryption technologies [14], on the impact of different cryptographic algorithms [15], or on specific types of devices [16], [17], [18]. There are also more holistic studies, like [19], where architectures including 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. After all, SPDM’s secure sessions are meant to protect in-transit messages, not only the actual data written to or read from the disk. Therefore, SPDM 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 that are already in encrypted form.

All in all, 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. To the best of our knowledge, this is the first work to assess the impact of SPDM’s security overhead on userspace metrics.

## 6 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 to increase the security level of modern computing systems, it is expected to bring performance penalties. Our goal in this paper is to assess the magnitude of SPDM’s performance impact. To the best of our knowledge, this is the first study that takes on this endeavor.

In summary, our results show that the overhead introduced by the most time-consuming SPDM message is 17.4ms (47.9 million cycles), while the fastest messages take only a few microseconds. According to our experiments, the typical SPDM bootstrap takes approximately 50ms to run. Regarding the hard drive benchmarks, we noticed that the specific workload greatly influences the final outcomes. On sequential read or write operations, data encryption becomes the bottleneck, and heavily affects performance.

On sequential read or write operations, data encryption becomes the bottleneck, and heavily affects performance (e.g., reading speed dropped from  $2.0\text{GB/s}$  to  $28\text{MB/s}$  in one run of the benchmark). That is not the case, though, 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, since the bottleneck is the physical movement of disk heads and switching between reading and writing modes.

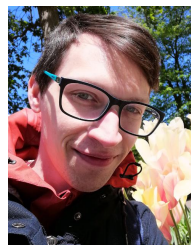
As future work, we intend to explore further the performance impacts of deploying SPDm in modern systems. This includes integrating SPDm with other classes of commonplace devices, such as network cards, and evaluating the protocol's overhead at earlier stages of the boot process.

## ACKNOWLEDGMENT

The authors would like to thank Hewlett Packard Enterprise for supporting this project. FAPESP 2020/09850-0.

## REFERENCES

- [1] A. Cui, M. Costello, and S. J. Stolfo, "When firmware modifications attack: A case study of embedded exploitation," in *Network and Distributed System Security Symposium (NDSS'13)*. The Internet Society, 2013.
- [2] D. Brown, T. Walker III, J. Blanco, R. Ives, H. Ngo, J. Shey, and R. Rakvic, "Detecting firmware modification on solid state drives via current draw analysis," *Computers & Security*, vol. 102, p. 102149, 2021.
- [3] B.-C. Choi, S.-H. Lee, J.-C. Na, and J.-H. Lee, "Secure firmware validation and update for consumer devices in home networking," *IEEE Transactions on Consumer Electronics*, vol. 62, no. 1, pp. 39–44, 2016.
- [4] J. Menn, "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: 2021-12-16.
- [5] DMTF, "DSP0274: Security protocol and data model (SPDM) specification, v.1.1.0," Distributed Management Task Force, Tech. Rep., Jul 2020, [www.dmtf.org/sites/default/files/standards/documents/DSP0274\\_1.1.0.pdf](http://www.dmtf.org/sites/default/files/standards/documents/DSP0274_1.1.0.pdf).
- [6] QEMU, "QEMU - a generic and open source machine emulator and virtualizer," <https://www.qemu.org/>.
- [7] J. Kukunas, "Chapter 8 - perf," in *Power and Performance*. Boston: Morgan Kaufmann, 2015, pp. 137–165. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/B9780128007266000082>
- [8] DMTF, "libspdm is a sample implementation that follows the DMTF SPDm specification," <https://github.com/DMTF/libspdm/>.
- [9] C. Profentzas, M. Günes, Y. Nikolakopoulos, O. Landsiedel, and M. Almgren, "Performance of secure boot in embedded systems," in *2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2019, pp. 198–204.
- [10] O. Khalid, C. Rolfes, and A. Ibing, "On implementing trusted boot for embedded systems," in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2013, pp. 75–80.
- [11] H. Yin, H. Dai, and Z. Jia, "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*, 2011, pp. 1188–1195.
- [12] V. B. Y. Kumar, N. Gupta, A. Chattopadhyay, M. Kasper, C. Krauß, and R. Niederhagen, "Post-quantum secure boot," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 1582–1585.
- [13] S. Dasari and V. Madipadga, "Aegis: A framework to detect compromised components in the supply chain of information technology infrastructure," in *2020 International Workshop on Big Data and Information Security (IWBIS)*, 2020, pp. 159–164.
- [14] A. Fujimoto, P. Peterson, and P. Reiher, "Comparing the power of full disk encryption alternatives," in *2012 International Green Computing Conference (IGCC)*. IEEE, 2012, pp. 1–6.
- [15] M. Brož, M. Patočka, and V. Matyáš, "Practical cryptographic data integrity protection with full disk encryption," in *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer, 2018, pp. 79–93.
- [16] M. Petschick, "Full disk encryption on unmanaged flash devices," Master's thesis, Technische Universität Berlin, Germany, 2011.
- [17] B. Bosen, "Full drive encryption with Samsung solid state drives," Trusted Strategies LLC, Tech. Rep., 2010.
- [18] —, "FDE performance comparison - hardware versus software full drive encryption," Trusted Strategies LLC, Tech. Rep., 2010.
- [19] R. Mayrhofer, "An architecture for secure mobile devices," *Security and Communication Networks*, vol. 8, no. 10, pp. 1958–1970, 2015. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1028>



**Renan C. A. Alves** received the B.Sc. degree in electrical engineering with emphasis on computer and digital systems, the M.Sc. degree, and PhD from the Universidade de São Paulo, Brazil, in 2011 and 2014, and 2020, respectively, where he is currently a post-doc and part-time professor. His main research interests include protocol modeling, performance analysis, internet of things, and cybersecurity.



**Bruno C. Albertini** Graduated in Computer Engineering from the Federal University of Rio Grande (2005), master's degree (2007) and PhD in Computer Science from the State University of Campinas (2012) in the area of Computer Architecture. He currently holds the position of Professor at the Department of Computer Engineering and Digital Systems (PCS) of the Polytechnic School of the University of São Paulo (EPUSP). Has experience in Computer Engineering, with emphasis on Hardware, working mainly on the following topics: hardware simulation, computational reflection, platform-based hardware design, computer architecture, wireless sensor networks, embedded systems, cryptohardware and AI in hardware. He joined LAA (Laboratory of Agricultural Automation) and BioComp in 2014, and has since applied his research to the environment, biodiversity and agriculture.



**Marcos A. Simplicio Jr.** is an Associate Professor at Escola Politécnica, Universidade de São Paulo (USP). He has a Master degree (2006) in Engineering conferred by the Ecole Centrale des Arts et Manufactures (Ecole Centrale Paris), France, and received his Computer Engineering PhD from USP in 2010. His main research interests are cryptography, cybersecurity, and distributed systems.