draft of 5-11-2018

# Self-encrypting deception: weaknesses in the encryption of solid state drives (SSDs)

Carlo Meijer
Radboud University, the Netherlands
C.Meijer@cs.ru.nl

Bernard van Gastel
Radboud University, the Netherlands
Open University of the Netherlands
Bernard.vanGastel@{ru.nl,ou.nl}

***Abstract*—We have analyzed the hardware full-disk encryption of several SSDs by reverse engineering their firmware. In theory, the security guarantees offered by hardware encryption are similar to or better than software implementations. In reality, we found that many hardware implementations have critical security weaknesses, for many models allowing for complete recovery of the data without knowledge of any secret.**

**BitLocker, the encryption software built into Microsoft Windows will rely exclusively on hardware full-disk encryption if the SSD advertises supported for it. Thus, for these drives, data protected by BitLocker is also compromised.**

**This challenges the view that hardware encryption is preferable over software encryption. We conclude that one should not rely solely on hardware encryption offered by SSDs.**

## I. Introduction

In recent years, protection of sensitive data has received increased attention. Protection of digital data has become a necessity, certainly in the light of new European Data Protection Regulation. Technically, encryption is the go to protection mechanism; it may be implemented in software or hardware (or both). It can be applied on the level of individual files, or the entire drive, which is called *full-disk encryption*. Full-disk encryption is often the solution of choice as it takes away concerns of sensitive data leakage through, for example, temporary files, page files and caches. Several software solutions for full-disk encryption exist, and modern operating systems typically integrate it as a feature. However, purely software-based encryption has inherent weaknesses, such as the encryption key being present in RAM at all times and performance drawbacks.

In an attempt to address these weaknesses, hardware full-disk encryption is often proposed; the encryption is performed within the drive itself, thereby confining the encryption key exclusively to the drive. Typically, the encryption itself is performed by a dedicated AES co-processor, whereas the software on the drive (firmware) takes care of the key management. It is often regarded as the successor of software full-disk encryption. Full-disk encryption software, especially those integrated in modern operating systems, may autonomously decide to rely solely on hardware encryption in case it is supported by the storage device (via the TCG Opal standard). In case the decision is made to rely on hardware encryption, software encryption is disabled. In fact, BitLocker, the full-disk encryption software built into Microsoft Windows, switches off software encryption and completely relies on hardware encryption by default if the drive advertises support.

*Contribution.* This paper evaluates both internal and external storage devices, from multiple vendors, adhering to standards for secure storage. The vendors combined produce close to half of the SSDs currently sold. An overview is given of possible flaws that apply in particular to hardware-based full-disk encryption (Section V), and a methodology is provided for the analysis (Section IV). We have analyzed firmwares from different SSD models offering hardware encryption, focusing on these flaws (see Section VI and Table I). The analysis uncovers a pattern of critical issues across vendors. For multiple models, it is possible to bypass the encryption entirely, allowing for a complete recovery of the data without any knowledge of passwords or keys. The situation is worsened by the delegation of encryption to the drive by BitLocker. Due to the default policy, many BitLocker users are unintentionally using hardware encryption, exposing them to the same threats. As such, we should reconsider whether hardware encryption is a true successor to its software counterpart, and whether the established standards actually promote sound implementations.

*Related work.* At OHM in 2013, Domburg demonstrated the possibility of debugging a hard drive through JTAG and created possibly the first publicly demonstrated hard drive firmware rootkit [4]. Domburg's work has inspired more research around anti-forensics such as [20], [7]. Leaked documents indicate that even the NSA is using these techniques [11]. Besides, proprietary cryptographic systems have often shown to be much weaker in practice than standardized publicly available alternatives once implementation details are uncovered [18]. Within the scope of storage devices with integrated hardware encryption, serious vulnerabilities have also previously been identified in external drives using proprietary protection schemes. An example is the external Secustick, which unlocks by simply sending a command (not containing a password) [5]. Another example is the Western Digital MyPassport family of external drives, which suffers from RAM leakage, weak key attacks, or even hardcoded keys [2]. However these findings are isolated incidents limited to proprietary solutions, and neither consider implementations of established standards for secure storage nor consider these issues across multiple vendors.

*Responsible disclosure.* After discovering these vulnerabilities, we followed a process of responsible disclosure. In this case, the National Cyber Security Center (NCSC) of the Netherlands was informed first, which assisted in the responsible disclosure process, by setting up the contact with the manufactures involved. We cooperated with both manufacturers, to fix their products and agreed not to disclose the vulnerabilities for six months. Both vendors have confirmed all the reported issues. For models currently being supported, firmware updates are either released or currently in development.

## II. Background

### A. Software vs Hardware Encryption

To avoid negatively impacting the data throughput when encryption is switched on, SSDs with encryption support, or *self-encrypting drives (SEDs)*, house a dedicated AES co-processor that provides for the encryption. Therefore, data encryption is essentially ‘free’ in terms of computational resources. These drives encrypt all data stored on them with the *disk encryption key* (DEK), even in case when the data is not password-protected. All drives considered in this paper use this approach. This essentially transforms the problem of protecting the data to protecting the DEK, introducing the following benefits:

(i) The data stored can be wiped instantly by erasing the DEK.
(ii) Setting or changing the password does not require re-encryption of all user data.

### B. Hardware encryption standards

*ATA Security:* The standard for ATA storage devices [17] defines the *security feature set*, which allows for locking and unlocking with a password. The goal of the ATA security feature set was limited to access control: it did not aim to stop a well-motivated attacker with physical access. At the time SEDs surfaced the market, it made sense to re-purpose the ATA security password for encryption. However, since the feature set already existed, it does not standardize cryptographic primitives, or even state that encryption should be used.

SED manufacturers commonly advertise that their products use strong cryptography, such as AES-256. Unfortunately, drive manufacturers typically do not provide encryption implementation details, or in case of ATA security, even state whether the encryption is used at all. In our opinion, it is reasonable to assume so. However, the standard is not violated in any way in case the password is used for access control alone. From the ATA standard [17]:

> If security is enabled on the device, the use of the Master password is indicated by the MASTER PASSWORD CAPABILITY bit. The MASTER PASSWORD CAPABILITY bit represents High or Maximum as described in this subclause. The MASTER PASSWORD CAPABILITY bit is modified during the processing of a SECURITY SET PASSWORD command that specifies a User password. If the MASTER PASSWORD CAPABILITY bit is set to High (i.e., zero), either the User password or Master password are used interchangeably. If the MASTER PASSWORD CAPABILITY bit is set to Maximum (i.e., one), the Master password is not used with the SECURITY DISABLE PASSWORD command and SECURITY UNLOCK command. The SECURITY ERASE UNIT command, however, uses either a valid User password or Master password.

By default, the Master password is set by the manufacturer. In case the user sets a password, he must take care to either also change the Master password, or set the MASTER PASSWORD CAPABILITY bit to Maximum. If he fails to do so, the Master password allows anyone with knowledge of the factory-default password to access his data.

*TCG Opal:* TCG Opal [9] is a newer specification for SEDs. It encompasses a communication protocol that is layered on top of ATA or NVMe. Furthermore, Opal mandates the use of either AES-128 or AES-256. The encryption should meet the bandwidth capability of the storage device. Opal compliant drives allow multiple passwords (*credentials* in Opal terminology) to be defined. Each can be assigned to perform various actions within the Opal subsystem. Special Admin credentials are used to perform provisioning and configuration.

A storage device can be divided into multiple *locking ranges*, that can be locked or unlocked independently. Each locking range is encrypted with a different DEK (*Media Encryption Key* in Opal terminology), and each locking range can be erased independently of the others. Cryptographic erase is performed by generating a new DEK. A special *global* range is defined as the range that covers all sectors of the disk not covered in other ranges.

Multiple passwords can be assigned permission to unlock a particular range. Additionally, a single password can be assigned permission to unlock multiple ranges. Phrased differently: a many-to-many relation exists between passwords and locking ranges.

*Proprietary alternatives:* Several proprietary alternatives exist. Examples are Seagate DriveTrust, the Western Digital MyPassport family of drives and Samsung’s portable SSDs. There are several reasons for manufacturers to prefer a proprietary solution over an open one. For example, the standard may have been introduced before Opal came into existence, or because a simpler scheme is preferred over Opal.

## III. Attacker model

Here we list several attacker models relevant in the context of full-disk encryption. In the rest of this article, we will only be concerned with the last one, as the implications of the first two are roughly equivalent when offsetting software against hardware encryption. We do, however, list them all here because it is in our opinion important to state why they are equivalent.

**Machine off, no awareness.** The adversary has momentary physical access to the powered-down machine, and the victim is unaware of this, creating an opportunity for the so-called *evil maid attack*. The encounter is used to install data exfiltration software or hardware on the victim’s machine.

In case of a hardware modification, e.g. a physical key logger device, to the best of our knowledge, no meaningful countermeasure exists today. For software modifications, the story is more nuanced. PCs fitted with a *Trusted Platform Module* (TPM) can take advantage of the *sealing* functionality, where cryptographic key material is bound to the software and hardware.

Hardware full-disk encryption does not mitigate the evil maid scenario in any meaningful way. Therefore, this attacker model is out of scope.

**Machine on.** The adversary has physical access to a powered-on machine while the encryption containers are unlocked. Software-based encryption solutions typically keep the cryptographic key in RAM, which is vulnerable to *cold boot attacks*, *DMA attacks*, or any other means of data exfiltration, including physical removal and readout with an external device. However, it is worth mentioning that software encryption exists that defends against such attacks, by storing the secret keys in CPU registers [12], [14].

An argument that is often put forward in favor of hardware encryption is that the secret key is not stored in RAM, and therefore is not vulnerable to the aforementioned attacks. In reality, this argument is invalid for several reasons.

(i) The software running on the host PC controlling the hardware encryption, typically *does* keep a secret key in RAM, introducing the same vulnerability. The reason is to support *Suspend-to-RAM* (S3), a low-power state wherein all peripheral devices are shut down. Since the SSD is powered down, it must be unlocked again once the system is resumed, and therefore either the operating system must retain a copy of the secret key at all times, or the user must enter it again. In virtually all implementations, including BitLocker, the former approach is chosen [13].

(ii) The burden of keeping the secret key is moved to the SSD, not eliminated. The SSD typically keeps the key in the main memory of its controller. SSDs are not security-hardened devices by any standard. In fact, many have a debugging interface exposed on their PCB, allowing one to attach a debugging device and extract the secret key from the drive. Furthermore, several means of obtaining code execution on the drive exist (See Section IV-B2).

(iii) A memory readout attack against software encryption requires physical access. Given this, the attacker also has the opportunity to carry out a hot-plugging attack against hardware encryption. This has been demonstrated in practice and poses a realistic threat [13].

As with the previous attacker model, opportunities and subsequent impact are roughly equivalent compared to software encryption. Therefore, this attacker model is also out of scope.

**Machine off, awareness.** The adversary has physical access to a powered-down machine, and the victim is aware of this. Therefore, from that point onward, the victim is unwilling to enter key information into the machine. In this scenario, given that the implementation is sound, software full-disk encryption offers full confidentiality of the data, and hardware encryption supposedly does so as well. In this paper, we focus on this attacker model.

## IV. Methodology

In order to assess how well the Opal standard performs in practice, we argue that we should analyze its implementations. This is, in our opinion, the most realistic measure. Such an analysis is inherently a somewhat ad-hoc process, since implementations vary wildly among manufacturers and models. However, to the extent possible, we document a generic approach that is applied to every device subject to analysis. In this remainder of section, we go through each step.

### A. Obtaining a firmware image

The difficulty of obtaining a firmware image from an SSD varies greatly among manufacturers and models. Below, we list a few examples.

*1) Downloading a firmware update:* Most manufacturers distribute firmware updates for their SSDs. Either by making them available for download from their website, or through their SSD management utility. For all the drives we studied, firmware updates consist of the entire firmware image.

Firmware updates downloaded from a manufacturer's website often comprise of a bootable ISO image, containing an operating system, firmware update utility, and the firmware image itself. The update utility applies the update using the 0x92 DOWNLOAD MICROCODE ATA command. Extracting the firmware from the ISO image is typically straightforward.

Obtaining a firmware image distributed through SSD management utility typically requires more effort, but is certainly not impossible. For example, the utility may apply obfuscation on its communication channels and/or firmware images that require some reverse engineering in order to remove. Furthermore, in case the target drive already has the latest version of the firmware installed, the utility may refuse to download the update, complicating the matter.

Some manufacturers use encrypted firmware images; the image is transferred to the drive, and subsequently decrypted by the drive itself. In this case, a means of low level control over the device, such as JTAG, or unsigned code execution, is required in order to extract the encryption key used. However, both means of control also allow us to simply extract the currently running firmware from RAM.

*2) Using a means of low level control:* Low level control over the device is valuable in itself, but, as stated above, can also be used to obtain a copy of the firmware, by extracting the currently running firmware from the device's RAM. Below we discuss several methods for gaining low level control.

### B. Gaining low level control over the device

A firmware image allows for static analysis. However, the possibility of dynamic analysis through e.g. JTAG is a significant advantage. It allows us to quickly confirm (or refute) assumptions and findings resulting from static analysis. Furthermore, in case weaknesses are found in the cryptographic scheme, a means of low level control is often required in order to exploit them.

*1) JTAG:* JTAG allows full control over a device. Through JTAG, we can halt/resume the CPU, read/modify registers and place break-points. Given these primitives, we can read/write arbitrarily in the address space, and execute arbitrary code. Some SSDs expose a JTAG debugging interface on their PCBs. Several standardized pin layouts exist. However, manufacturers may opt for a proprietary one. The JTAGulator[8] allows us to automatically determine whether a set of pins speak the JTAG protocol, and if so, the purpose of each pin.

*2) Unsigned code execution:* Some SSD manufacturers prefer to restrict what their end users can do with their devices. Hence, they typically disable the JTAG feature of the storage controller. In the absence of JTAG, a suitable alternative is the ability to execute arbitrary code on the storage controller, as it allows for essentially the same capabilities. However, all drives in our study have countermeasures in place to prevent this, such as cryptographic signature verification of firmware updates.

Still, various means of gaining code execution exist:

(i) Use a vendor-specific undocumented command, if present.
(ii) Exploit a vulnerability in the firmware, typically involving memory corruption.
(iii) Communicate directly with the drive's memory chips with an external reader device, and modify the currently installed firmware.
(iv) Perform a fault injection attack (power, electro-magnetic, or otherwise) in order to trick the drive into accepting a modified firmware update with an invalid signature.

**Vendor-specific commands** Most manufacturers implement vendor-specific commands for information gathering, diagnostics, and other purposes. Through static analysis of firmware images, we found examples in which a command exists that allows for arbitrary values to be written to a memory address of choice. This can be leveraged into arbitrary code execution, e.g. by overwriting a function pointer.

**Memory corruption** The extent to which manufacturers attempt to prevent memory corruption vulnerabilities varies. Therefore, the success rate for identifying such a vulnerability does so as well. Memory corruption vulnerabilities can in many situations be leveraged into unsigned code execution, a stack-based buffer overflow is an example of this.

**Storage chip communication** A more invasive technique for gaining unsigned code execution is by using an external reader device to make modifications to the currently installed firmware. We make a distinction between NAND and NOR memory.

The NAND flash chips usually contain the user-accessible storage. Retrieving data from a NAND chip requires one to know several chip characteristics, such as the page, block and plane size. Some NAND chips have proprietary extensions, e.g. in order to allow for assigning a region as SLC memory. Furthermore, NAND chips in SSDs typically come as BGA packages, requiring them to be desoldered from the PCB before they can be directly communicated with.

Alternatively, some SSDs also house a NOR flash, connected through SPI, a simple and well-supported protocol. SPI flash chips usually expose their pins on the outside, allowing for direct communication without the need for desoldering. The purpose of this NOR flash varies. Typically, it contains the drive's capacity, serial number, NAND chip characteristics, error logs, and more. In some occasions, it contains executable code. In such a case, unsigned code execution may be possible by making modifications to that executable code.

**Fault injection attacks** Finally, although we have not attempted it during any of our case studies, a fault injection attack may be used to obtain unsigned code execution. For example, by triggering a clock glitch during a firmware update, causing a conditional jump instruction to be skipped, tricking the drive into accepting a firmware update with an invalid cryptographic signature. In order to successfully mitigate fault injection attacks, both hardware and software countermeasures are necessary. Neither of which any of the drives we investigated has in place. Moreover, to the best of our knowledge, no SSD controller on the market exists today that has hardware countermeasures against fault injection attacks. Hence, fault injection is likely a means of gaining unsigned code execution on SSDs now and in the foreseeable future.

## C. Analyzing the firmware

Once a firmware image for a particular drive is acquired, we analyze it. The file format used for firmware images differs between manufacturers, and occasionally between different models from the same manufacturer. For all drives we studied, the image is divided in sections. Essential information about these sections, such as their size, memory address, and offset in the file, is usually contained within the image header. This information is important for the analysis, since the firmware code may at times refer to resources via absolute memory addresses. In some cases, the section information is immediately apparent by inspection. In other cases, some reverse engineering of the code responsible for interpreting firmware images is needed.

Once the sector information is uncovered, the firmware image can be loaded into a disassembler and analysis tool. We used the IDA Pro for this purpose.

When reverse engineering SSD firmwares, a good starting point is identifying the *ATA dispatch table*, i.e. an array of data structures containing the ATA opcode, the address of the function that implements it, and possibly other data. All drives in our study implement the ATA standard in a way similar to this. Once the table is identified, the implementation of any desired command can be studied by analyzing the code located at the respective address.

For each of the possible issues given in Section V, we attempt to find out whether the drive is susceptible to it by studying the relevant code. Note that analyzing this many drive firmwares for this many weaknesses is considerably time-consuming. Therefore, it is in our opinion justified that we skip the analysis of the other issues listed in Section V, once we identify an issue that fully compromises the drive's encryption, since it does not contribute to the final assessment of the security for that particular drive.

## V. Possible Security Issues with Hardware Encryption

We argue that full disk encryption should be implemented carefully by experts, and should be under public scrutiny whenever possible. Properly implementing a hardware FDE scheme is not trivial. To substantiate this claim we pose a number possible implementation pitfalls. The list presented in the remainder of this section is used as a guideline in Section VI in order to assess how well hardware encryption is generally implemented.

### A. Password and DEK not linked

Obviously, the password should be required in order to obtain the DEK, and this requirement should be cryptographically enforced. Absence of this property is catastrophic. Indeed, the protection of the user data then no longer depends on secrets. All the information required to recover the user data is stored on the drive itself and can be retrieved.

Unfortunately, implementing this properly is not entirely trivial. As stated in Section II-B, standards dictate that multiple passwords yield the same DEK and that passwords can be changed independently.

### B. Single DEK used for the entire disk

The Opal standard allows for multiple ranges to be defined, each of which protected with different passwords. A naïve implementation is to use a single DEK for the entire drive, and store an encrypted variant of it for each password, whereas a proper implementation produces different DEKs for each range.

On the surface, doing so may seem only a minor issue. Indeed, access to at least one range is still required. However, the (probably) most popular Opal management software, BitLocker, leaves the global range unprotected in order to allow the partition table to be accessible. Consequently, the DEK must be stored unprotected to allow for this, in effect compromising the other ranges.

### C. Lack of entropy in randomly generated DEKs

Within the ATA and Opal standards, no means exist for the end user to specify the DEK himself. The only way to affect its value is by randomizing it. This raises the question whether sufficient random entropy is available during the DEK generation.

In principle, the environment wherein SSDs are deployed allows for sufficient entropy to be acquired. For example, the drive's temperature sensor and I/O requests from the host PC. Furthermore, storing and restoring the random pool upon reboots should not be an issue since we are concerned with storage devices. However, random number generators in embedded devices have a notoriously bad reputation [19].

### D. Wear leveling

SSDs use flash memory for data storage. A property of flash memory is that it can be put through a limited number of write-erase-cycles before becoming unreliable. In order to prolong the service life of the device, *wear leveling* is applied. It works by arranging data so that erasures and re-writes are evenly distributed across the medium. This way, no single block prematurely fails due to a high concentration of write cycles. Thus, multiple writes to the same logical sector typically trigger writes to different physical sectors. Older copies of a sector remain stored until overwritten (although not directly retrievable by the end user).

This raises the question whether this applies to key information as well. Suppose that the DEK is stored unprotected, after which a password is set by the end user, replacing the unprotected DEK with an encrypted variant. Due to wear leveling, the new variant can be stored somewhere else within the storage chip and the old location is marked as unused. If not overwritten later by other operations, the unprotected variant of the DEK can still be retrieved.

### E. Power-saving mode: DEVSLP

DEVSLP is a feature that allow SATA drives to go in to a low power 'device sleep' mode when sent the appropriate signal. The advantage over other modes is that the SATA link need not be powered to receive a wake-up trigger. Instead, an out-of-band signal is sent over the rarely used and now obsolete 3.3V pins of the SATA power plug.

How much power is consumed when the drive is in DEVSLP depends on the implementation. The ATA standard is not explicit about how the power consumption reduction is to be achieved. A manufacturer may freely choose, for example, to have the drive write its internal state to non-volatile storage and subsequently power down the RAM. The drive complies to the standard as long as the it can become operational within 20ms of receiving the wake-up signal.

Suppose that a drive indeed writes its internal state to non-volatile memory. Then care must be taken that the state from non-volatile memory is erased upon wake-up, or else an attacker may be able to extract the DEK from the last stored state.

### F. General Implementation Issues

All the issues depicted above in this section apply in particular to hardware-based disk encryption. However, potential implementation issues in software-based encryption may also apply. Examples include re-use of the initialization vectors and using an insecure mode of operation.

Choosing the right mode of operation and implementing it correctly can be tricky, as the chosen mode must allow for both random read and write access, not allow for exchange of ciphertexts, and not be malleable. Many software-based solution, such as VeraCrypt and later versions of Microsoft BitLocker, use the XTS mode of operation. A description of XTS is given below.

The XTS, or *XEX Tweakable Block Cipher with Ciphertext Stealing* [1], mode of operation was designed for cryptographic protection of data on storage devices of fixed length data units. It is an instantiation of Rogaway's XEX (XOR Encrypt XOR) tweakable block cipher [16], extended with ciphertext stealing to support arbitrary length inputs. Furthermore, XEX mode uses a single key for both encryption and tweaking, whereas XTS mode uses two independent keys.

XTS mode provides confidentiality for the protected data. Authentication is not provided, because one of the design goals is to provide encryption without data expansion. In the absence of authentication or access control, the best one can do is to ensure that any alteration of the ciphertext will completely randomize the plaintext, and rely on the application that uses this transform to include sufficient redundancy in its plaintext to detect and discard such random plaintexts. In light of this, XTS provides more protection than other confidentiality-only modes against manipulation of the encrypted data.

The XTS mode of operation has received criticism [15], [3]. An important point is that the granularity to which an attacker has the ability to randomize plaintexts must equal the cipher's block size. In case of AES, this is 16 bytes. Ferguson has designed a native diffuser function that addresses this problem for application in BitLocker [6]. In the same publication, XTS is not mentioned, but LRW mode with the same limitation is criticized likewise.

## VI. Case Studies

### A. Crucial MX100

The Crucial (Micron) MX100 is a SATA SSD released in 2014. It features MLC NAND flash memory. It supports ATA security, as well as TCG Opal, both version 1 and 2. The controller used in the MX100 is the Marvell 88SS9189. It houses a dual-core 88FR102 V5 (ARM) CPU. At the time of release, its performance is close to that of the competition, although the drive is considerably less costly.

*Firmware:* A firmware update is available for download through Micron's website. It comes as a Linux-based bootable ISO image. The firmware image is stored within the ISO image, and is sent unmodified to the drive through the ATA 0X92 DOWNLOAD MICROCODE command. From this point onward, the drive takes care of the firmware update process.

The firmware image is cryptographically signed using 2048-bits RSA and SHA256. The signature verification is based on mbedTLS's `rsa_pkcs1_verify` function.

Fig. 1. JTAG pins on the Crucial MX100.

*Debugging:* The MX100 has a JTAG interface that can be used to connect a debugger device. The location on the PCB is depicted in Figure 1. The standardized ARM14 JTAG pin layout is used.

*Findings:* In this section, we present our findings with respect to the MX100. It covers both ATA security and TCG Opal implementation. Furthermore, we discovered several proprietary vendor commands.

**ATA security.** We found that the implementation of the ATA `0xf2 security unlock` command passes the incoming password to the SHA256 hash function. Subsequently, the output is compared to another buffer. If the comparison succeeds, the drive unlocks. However, the original password buffer remains unused during this process. Hence, cryptographic binding between password and DEK is lacking.

**TCG Opal.** We discovered that the TCG Opal implementation works in a similar fashion; i.e. no cryptographic binding between password and DEK is present.

**Vendor-unique commands.** The MX100 features several vendor-specific commands that allow engineers to interact with the device. By default, the commands are restricted: they must be unlocked before they can be used. The list presented below is far from exhaustive.

*Unlocking.* Unlocking the vendor-specific features is done by issuing a `0xfd` (vendor-specific) ATA command, with feature code `0x55`. Setting the LBA to `0x306775`, and the *block count* to `0x65` will unlock the vendor-specific commands.

*Reading a page from NOR flash.* The NOR flash stores various data, among them is the device capacity, serial number, error logs, and boot loader (the boot process is identical to that of the MX200 and MX300, see Section VI-C). A page can be retrieved with the `0xfa` (vendor-specific) ATA command, with feature code `0xd2`. The LBA is the page number that is to be retrieved. A NOR page is always 128 KB.

*Erasing a page in NOR flash.* A NOR page can be erased with the `0xfc` (vendor-specific) ATA command, with feature code `0xe2`. The LBA is page number. No data is transferred.

*Writing to a page in NOR flash.* A NOR page can be written to with the `0xfb` (vendor-specific) ATA command, with feature code `0xd2`. The LBA is once again set to the page number. The transfer size should be 128 KB.

*Arbitrary memory write.* The MX100 has a command that allows one to write arbitrary data to any desired address within the address space. The command listens to opcode `0xfb` (vendor-specific) and feature code `0x23`. The command expects a concatenated list of address-value tuples.

*Security evaluation:* The MX100 has critical security issues in both the ATA security and TCG Opal implementation. Namely, no cryptographic binding is present between password and DEK. The scheme is essentially equivalent to no encryption, as the encryption key does not depend on secrets. We demonstrated in practice that, by modifying the password validation routine in RAM through JTAG, the MX100 unlocks with any password, and the drive's contents become accessible. This applies to both ATA security and TCG Opal. We have not studied the MX100 for the other weaknesses described in Section V, since the drive's encryption is already compromised.

Furthermore, we found that a vendor-specific command allow for arbitrary modifications within the address space. This enables malware with remote access to the host PC to infect the drive's firmware, allowing it to hide itself and/or to survive re-installation of the host PC's OS.

*Attack strategy:* Suppose that we want to recover the data from a locked MX100 drive for which we do not have a valid password. In order to do so, we connect a JTAG debugging device to the pins depicted in Figure 1. Subsequently, we use it to modify the password validation routine in RAM so that it always validates successfully, regardless of the input password. Finally, we unlock the drive as normal, with an arbitrary password. The strategy is the same for both ATA security and TCG Opal.

## B. Crucial MX200

The Crucial MX200 is a SATA SSD released in 2015. It is essentially an MX100 with an SLC write cache. The MX200 is built around the same 88SS9189 controller. The firmware is very similar to that of the MX100. Due to the similarities, the analysis is limited to verifying whether the same vulnerabilities are present.

*Security evaluation:* We found that the MX200 suffers from the same lacks of cryptographic binding between password and DEK. This applies to both ATA security and TCG Opal. In both cases, we were able to demonstrate in practice that the encryption can be completely bypassed by modifying the password validation routine through JTAG.

Furthermore, the vendor-specific commands found in the MX100 are also present in the MX200. As such, a remote attacker is able to gain code execution on the device.

*Attack strategy:* The attack strategy is identical to that of the MX100. See Section VI-A.

## C. Crucial MX300

The Crucial MX300 is a SATA SSD released in 2016. It is the successor to the MX200. Since the MX300, a switch has been made to TLC memory. Similar to both its predecessors, it supports the ATA security feature set, as well as TCG Opal version 1 and 2. The MX300 is fitted with a Marvell 88SS1074 controller, the successor to the 88SS9189. The MX300's firmware differs from its predecessors in some aspects, including the JTAG feature being switched off (although supported by the controller), and the code related to cryptography being subject to a major revision.

*Debugging:* A firmware image can be obtained through Micron's website. It uses the same file format as that of its predecessors. Hence, the firmware can be analyzed. As stated in Section IV-A2, JTAG allows for low level monitoring and control of the storage controller's CPU. It significantly aids the analysis, as it allows for verification of assumptions and findings, and possibly exploitation of weaknesses. Hence, absence of this feature is problematic. Therefore, we used the strategies listed in Section IV-B2 in order to acquire unsigned code execution on the device.

We found that the vendor-specific commands present in the MX100 and MX200 that allow us to gain unsigned code execution, are still present. However, since the MX300, the unlock command for vendor-specific commands is deprecated and replaced by another that relies on asymmetric cryptographic signatures. Hence, the vendor commands no longer serve as a vehicle for unsigned code execution. Furthermore, we identified several memory corruption vulnerabilities. None of which we could successfully exploit in order to gain control over the execution.

Below we describe how we acquired unsigned code execution by directly communicating with the drive's NOR flash.

*Findings:* **Obtaining unsigned code execution** As stated previously, we used an external reader device to communicate with the NOR flash through SPI, allowing its contents to be retrieved and manipulated. In order to leverage this into unsigned code execution, we must first understand the device's boot process, which we reverse engineered. A diagram depicting the boot process is given in Figure 2.

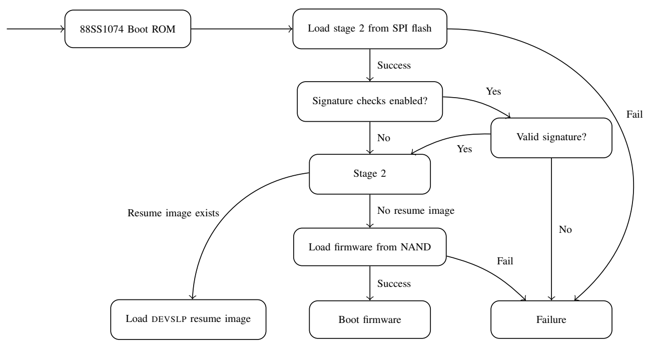

Fig. 2. Crucial MX300 boot process.

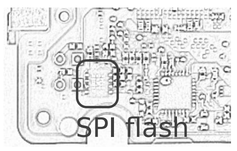

Fig. 3. SPI flash chip on the MX300 PCB.

Once the storage controller is powered on, the first instructions executed by its CPU are stored in a ROM, embedded in the controller. The ROM code loads its next boot stage from the SPI flash. It is located on the PCB as indicated in Figure 3. We refer to the code stored in the SPI flash as *stage 2*. It is responsible for, among other things, initialization of the NAND flash and DRAM memory. Subsequently, it retrieves the drive's firmware from NAND and copies it to DRAM. Then, after a number of integrity checks, it transfers control to the firmware.

We found that, in fact, the 88SS1074 controller supports cryptographic signature verification of stage 2. However, the MX300 does not take advantage of this feature. Hence, one can freely make modifications to the stage 2 code by modifying the contents of the SPI flash with an external reader device.

Ideally, we would like to have the capability of arbitrarily retrieving and modifying code and data while the firmware is running. Given these primitives, unsigned code execution is possible as well. For example by using the write primitive to overwrite a non-critical ATA command handler function with the desired code and subsequently issuing the corresponding command. We created a modified firmware image, which includes these arbitrary read/write capabilities.

In order to convince the drive to accept the modified firmware image, the cryptographic signature checks during firmware updates need be bypassed. We accomplished this by modifying stage 2, injecting a piece of code that modifies the cryptographic signature verification function such that it accepts invalid signatures. The injected code runs directly after the firmware has been copied into RAM, and before transferring control to it. After power-cycling the drive, it accepts firmware images with invalid signatures and hence our modified firmware image can be sent to the drive as a firmware update. Once the modified firmware is installed, we have arbitrary read/write capabilities, and thus unsigned code execution.

**Key derivation scheme** We have reverse engineered the full-disk encryption implementation of the MX300. Its key derivation scheme is depicted in Figure 4. Compared to the MX100 and MX200, the full-disk encryption implementation differs significantly. Notably, cryptographic binding between password and DEK is introduced.

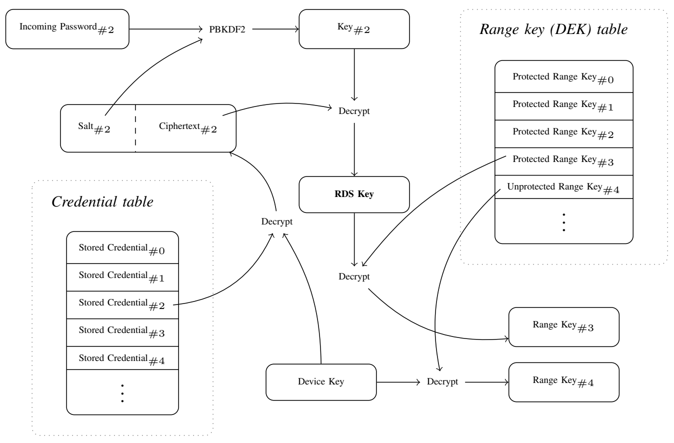

Fig. 4. Scheme used to obtain a range key (DEK) from the user-supplied password. In this example, credential #2 is used to unlock range #3.

Each MX300 drive has a per-device unique key, which we refer to as the *device key*. It is stored within a small chunk of non-volatile memory contained within the controller. As such, an attacker cannot obtain it, unless he has the ability to execute unsigned code on the controller's CPU.

As is mandated by TCG Opal, the scheme allows for multiple credentials and ranges. Each credential has an associated data structure stored within the NOR flash. This is what we refer to as the *credential table*. Entries within this table are encrypted using the device key. Each entry contains a *salt* and a *ciphertext*. The random salt and the user-supplied password are fed to PBKDF2. Subsequently, the result is used as a key in an attempt to decrypt the ciphertext. If the password is correct, the decryption succeeds, and the decrypted result is the so-called *RDS key* (referred to by the firmware as such). All stored credentials contain an encrypted version of the same RDS key, or a zero buffer, depending on the parameters used during the creation of the data structure.

Each locking range is protected with its own unique DEK. The DEKs are stored in the, what we refer to as, *range key table*. All keys corresponding to protected ranges (i.e. requiring a password before becoming accessible) are encrypted using the RDS key. All other DEKs are encrypted using the device key and are therefore always accessible.

From the description given above, we can already see that the RDS key can be obtained once only a single password is known. Subsequently, the RDS key allows access to all protected ranges. The drive will refuse to unlock a range for a user who does not have permission to access it. However, this check is not cryptographically enforced. This is already a weakness in the design of the key derivation scheme. However, we found that even a single password need not be known, which we explain in detail below.

**Opal Setup** During the set-up phase of TCG Opal, the credential table and range key table are populated. In order to better understand this process, we used our arbitrary write capabilities to inject tracing functionality at various places in the firmware. The execution trace generated during the BitLocker set-up phase is given in Appendix A. In case the full-disk encryption is set up using `sedutil`, an open-source utility for TCG Opal, the result is similar. Pseudocode for some of the routines captured is given in Appendix B.

From the execution trace, we can clearly see that once the BitLocker set-up phase is completed, the RDS key is protected (encrypted) with a zero buffer as a password, and stored in all slots between 11 and 29, with the exception of slot 15.

Hence, the RDS key can be recovered from any of these slots, by invoking the VerifyPasswd function with a zero buffer as password. This can be accomplished by means of unsigned code execution (Section VI-C). The result is that all DEKs can be decrypted without a password.

**ATA security** Suppose that the drive is protected by means of the ATA security feature set, rather than Opal. The key derivation scheme is essentially equivalent, with some differences:

(i) The range key table has a single entry: the DEK for the entire drive.
(ii) The credential table has two entries: the ATA User and Master password.
(iii) The RDS key used for protecting DEKs under Opal differs from the one used under ATA. Hence, recovering the RDS key as described above will work, but results in an RDS key that cannot successfully decrypt the DEK.

As stated in Section II-B, the MASTER PASSWORD CAPABILITY bit determines whether the factory-set Master password may unlock the drive. In order for the end user to redeem himself from the Master password acting as a security bypass mechanism, he either has to set the MASTER PASSWORD CAPABILITY bit to Maximum, or change the Master password.

In the case of the MX300, the former approach is insufficient. We found that the Master password allows for successful decryption of the RDS key, regardless of the MASTER PASSWORD CAPABILITY bit. Hence, in case the end user has set it to Maximum, but has not changed the Master password, the drive's contents is still accessible to anyone in possession of the default Master password. In the case of the MX300, this is an empty string.

In order to exploit this vulnerability, we only need to change the MASTER PASSWORD CAPABILITY bit located in RAM. This can be accomplished through the arbitrary write capability obtained previously. Once accomplished, the drive successfully unlocks as normal, using an empty string as the Master password.

*Attack strategy:* Suppose that we want to recover the data from a locked MX300 drive for which we do not have a valid password. In order to do so, we first install a modified firmware that includes arbitrary read/write capabilities. The process is described in detail in Section VI-C. The following steps describe how to recover the data from a drive that is set up through TCG Opal, or ATA security, respectively.

**TCG Opal** Once the custom firmware is installed, we use its arbitrary write capability in order to write executable code in the device's address space. The code is crafted such that it invokes the VerifyPasswd function with a zero buffer as password, using credential slot 11 and with bExtractRdsKey set to **true**. It should overwrite an existing non-critical ATA command handler function, for example, the SMART command handler. Issuing the corresponding ATA command then executes the code. At this point, the RDS key is extracted and copied to the global RDS key buffer and all protected range keys can be decrypted.

By using the arbitrary write capability once more, we modify the VerifyPasswd function such that it always returns SUCCESS. Note that this also implies that the function will no longer affect the global RDS key buffer, which is desired behavior since it already contains the correct RDS key. At this point, any password can be used to 'authenticate' successfully. We use `sedutil`, an open-source TCG Opal utility, to authenticate with an arbitrary password, and subsequently unlock any desired range. Note that we should choose a user that has permission to access that particular range. In case of BitLocker, USER2 may access range #1, although we can simply try all possible users. By default, `sedutil` authenticates as ADMIN1. However, it is trivial to modify its source code so that it authenticates as any other user.

**ATA security** We use the arbitrary write capability in order to change the MASTER PASSWORD CAPABILITY bit in RAM from Max (1) to High (0). Then, we authenticate to the drive as normal, using an empty string as the Master password, and unlock the drive.

Note that this approach will not work in case ATA security is used instead of Opal, with the Master password changed rather than disabled. However, we argue that this is an unlikely scenario.

## D. Samsung 840 EVO

The Samsung 840 EVO is a SATA SSD released in 2013. It features TLC NAND memory, which is more cost-effective than MLC and SLC. The 840 EVO features an SLC write cache. It is connected through SATA. It supports ATA security, as well as TCG Opal version 2. At its core is Samsung's own MEX controller, based on a triple-core Cortex R4 design. The 840 EVO boasts hardware AES-256 encryption.

*Firmware:* A firmware update can be downloaded through Samsung's website. It comes as a bootable ISO image. The firmware image is stored within the ISO image, albeit in an obfuscated form. De-obfuscation is performed by the update utility itself. Hence, recovery of the obfuscation algorithm is straightforward. The obfuscation algorithm has been previously reverse engineered[1].

Once the image is de-obfuscated, it is transferred to the drive using the ATA `0x92` DOWNLOAD MICROCODE opcode. From this point onward, the firmware update process takes place on the drive itself.

The firmware image is cryptographically signed with ECDSA. The curve and its exact parameters are yet to be determined. The hash function used is SHA256.

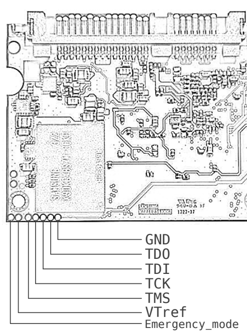

Fig. 5. JTAG pins on the Samsung 840 EVO.

*Debugging:* The 840 EVO has a JTAG interface. The pin layout is shown in Figure 5. The pin layout was found with help of the JTAGulator[8]. It was independently found by [10].

*Findings:* In this section we take a deep dive in the 840 EVO's encryption internals. Firstly, we present the full key derivation scheme from password to DEK. Secondly, we describe several proprietary vendor commands.

**Key derivation scheme** Before exploring the key derivation scheme, we first describe some of the data structures where the scheme is built upon.

*Key storage* The, what we refer to as, *key storage* data structure is depicted in Figure 6. The purpose is both password validation and key derivation. A password candidate $p'$ is validated by computing $\text{PBKDF2}(\text{HMAC\_SHA256}, p', s_{\text{verif}})$. Then, the result is compared against the stored $\text{PBKDF2}(\text{HMAC\_SHA256}, p, s_{\text{verif}})$. If they match, then $p = p'$ is assumed. The derived encryption key is then obtained by computing $\text{PBKDF2}(\text{HMAC\_SHA256}, p', s_{\text{deriv}})$.

*Crypto blob* All information related to disk encryption is stored in a single, 64 KB binary blob, which we refer to as the *crypto blob*. An overview of it is given in Figure 7. The crypto blob is divided in 128-byte slots, totaling to 512 slots. The purpose of each slot is determined by its slot number.

Every slot beyond number 451 is encrypted with AES-256 in XTS mode, with a key derived from known data. The purpose seems to be obfuscation. The key is $\text{HMAC\_SHA256}(p, s)$, where salt $s$ is `"bongbong.kim@samsung.com"` (padded to 32 bytes), and password $p$ is a permutation of the value stored in slot

[1]URL: https://github.com/ddcc/drive_firmware

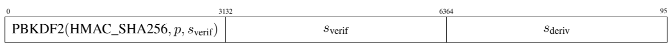

Fig. 6. Key storage data structure for password $p$.

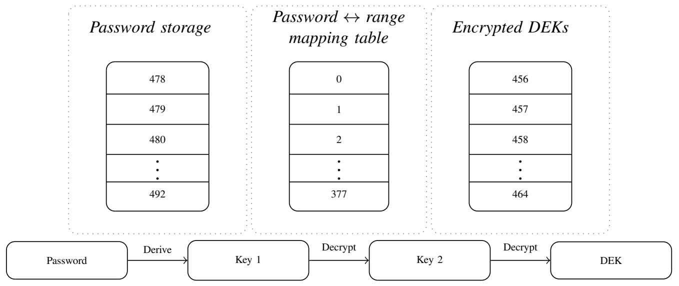

Fig. 8. Relation between passwords and DEKs.

451. The key is computed during the drive's boot sequence. However, due to a bug in the firmware, retrieving slot 451 during early boot fails. Therefore, $p$ is in fact a zero buffer. Consequently, the resulting key is constant for all devices.

Furthermore, the slot number is used as the IV for the XTS mode. However, due to a bug in either the management code, or in the AES co-processor itself, the IV value is ignored and zero is used instead for all slots. Due to this re-use of IV, two slots that share a plaintext block at some offset, will have the same ciphertext for that block.

As the purpose of the encryption is presumably obfuscation, confidentiality of user data is not affected.

*From password to DEK* Samsung's Opal implementation allows a total number of 9 ranges and 14 passwords to be specified. The password is fed to the validation/derivation function, using slot $478+u$, where $u$ is the user id associated with the password. Once the password has been validated, the derived key is then used to decrypt an entry in the password $\leftrightarrow$ range mapping table. The entry is located at slot $0 + 27u + 3r$, where $u$ is the user id and $r$ is the range number. Finally, the decrypted result is used to decrypt slot $456+r$, yielding the DEK for range $r$. A diagram picturing the process is given in Figure 8. The derived key also successfully decrypts slots $1 + 27u + 3r$ and $2 + 27u + 3r$. The purpose of these slots remains to be researched.

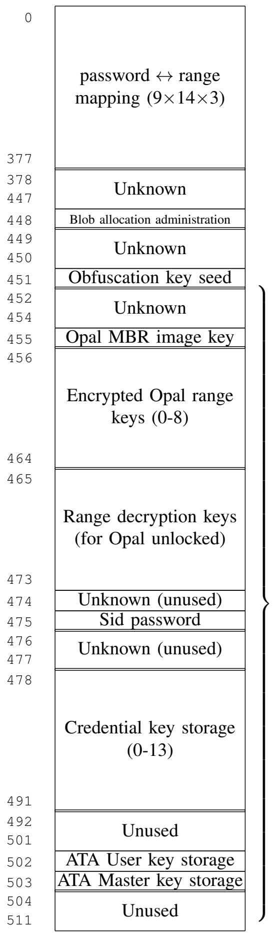

Fig. 7. Crypto blob.

**Vendor-unique commands** The 840 EVO features several vendor-specific commands. As is the case with the Crucial drives, these commands require unlocking. Only a small subset of commands are analyzed, since the vast majority have no relation with security.

*Unlocking.* Unlocking the vendor-specific features is done by issuing a `0x85` (vendor-specific) ATA command with feature code `0x46`. The payload is a single block (512 bytes) with the last 16 bytes set to

`C7D0B1B3C1BEC0CCB6AFB6AFBEEEBCAD`

*Retrieving the crypto blob.* The crypto blob can be retrieved by issuing a `0x83` (vendor-specific) ATA command, with feature code `0x12`. The output is the 64 KB crypto blob.

*Storing the crypto blob* The crypto blob can be stored by issuing a `0x83` (vendor-specific) ATA command, with feature code `0x13`. The handler expects a 64 KB input, which is then instantiated as the device's crypto blob.

*Security evaluation:* We managed to identify several implementation mistakes. Two of which, depending on the circumstances, can be leveraged into full recovery of the data.

**ATA security** The ATA password *may* be cryptographically bound to the DEK. This depends on the value of the MASTER PASSWORD CAPABILITY bit during the ATA security setup. Slots 502 and 503 of the crypto blob contain key storage data structures (Figure 6) for the User and Master password, respectively, allowing for password validation and key derivation. The DEK is always stored encrypted in slot 451. In case the MASTER PASSWORD CAPABILITY bit is set to Maximum, the decryption key is derived from slot 502. Otherwise, the key is stored in plain-text format in slot 465, and slot 502 and 503 are used for password validation only. Thus, allowing the encryption to be bypassed. We successfully demonstrated in practice that, by crippling the password validation routine in RAM, the drive successfully unlocks with any password.

**TCG Opal** After reverse engineering and carefully studying the design of the key derivation scheme used in Samsung's TCG Opal implementation, we have not identified any weaknesses.

**Random entropy** The 840 EVO has a hardware RNG. Although in many situations, a pseudo RNG is used, which works by encrypting an incrementing counter using the AES co-processor. The key is generated at startup by the hardware RNG. All key material is generated by the hardware RNG.

The hardware RNG passes all FIPS 140-2 tests over 1 Megabyte of random data. In theory this does not prove absence of weaknesses in the RNG. However, reverse engineering the hardware RNG is far beyond the scope of this research. We therefore assume that the output generated by the hardware RNG is cryptographically secure.

**Wear leveling** The Samsung 840 EVO stores its crypto blob on the device's NAND flash, albeit within a region designated for internal data structures. Despite this, the crypto blob storage is wear-leveled.

Suppose that at time $t_0$, the drive is in an unprotected state, i.e. neither ATA security nor TCG Opal is set up. In this state, the drive has a single locking range defined that covers the entire user-accessible storage. The DEK for this range is stored encrypted in slot 456. The decryption key is stored in slot 465. At time $t_0$, the crypto blob is stored at physical sector $s_0$ in flash. Subsequently, at time $t_1$, a password is set, either through ATA security, with the MASTER PASSWORD CAPABILITY bit set to Maximum, or through TCG Opal. As such, the DEK cannot be extracted from the crypto blob without a password. The updated crypto blob is stored at sector $s_1$ in flash.

Due to the wear leveling mechanism, $s_0 = s_1$ is not guaranteed. Therefore, at time $t_1$, the DEK may be recoverable by retrieving the crypto blob stored at physical sector $s_0$. We have successfully demonstrated this attack in practice. Once a previous revision of the crypto blob has been recovered, it can be instantiated through a vendor-specific command (see VI-D).

Empirical measurements indicate that $s_0 \neq s_1$ occurs approximately 1 in every 20 times the crypto blob stored (i.e. every time crypto related information is updated).

**Power-saving mode: DEVSLP** Support for DEVSLP is only present on the mSATA variant of the drive. Although functionality related to DEVSLP is clearly also present in the SATA variant's firmware, we were unable to trigger it.

**Mode of operation** User-accessible data regions are encrypted using AES-256 in XTS mode. The key is obtained through the key derivation scheme described in Section VI-D. The IV used is obtained by taking $b+n$, where $b$ is a base value which is generated randomly during the drive's initialization phase, and $n$ is the *Logical Block Address* (LBA) of the sector of the data that is to be encrypted.

From the description presented here, we conclude that no mode of operation related security issues are present in the 840 EVO, apart from XTS's inherent weaknesses (Section V-F).

*Attack strategy:* Suppose that we want to recover the data from a locked 840 EVO drive. The approach taken depends on whether the drive is protected with the ATA security feature set, with the MASTER PASSWORD CAPABILITY bit set to High.

If this is indeed the case, then the DEK is not cryptographically bound to the password. Hence, the only barrier we have to overcome is the password validation routine. We connect a JTAG debugging device to the pins depicted in Figure 5. Through JTAG, we modify the password validation routine such that it always validates successfully. Finally, we unlock the drive as normal, with an arbitrary password.

If ATA security, with the MASTER PASSWORD CAPABILITY bit set to Maximum, or TCG Opal is used, then the DEK is cryptographically bound to the password. However, due to the wear-leveling issue pointed out in Section VI-D, the data on the drive may still be recoverable by reverting to a previous version of the crypto blob that was used while the drive was in an unprotected state.

In order to do this, first, we craft code that searches the raw NAND flash for crypto blobs, at the region designated for internal data structures. Crypto blobs can be easily identified, as slot 450 always starts with the string `"secu0.01clas"`. We once again connect a JTAG debugging device, load the code into the device's address space and execute it. In case the drive is in an unprotected state, slot 465 contains the decryption key for the DEK at slot 456. Hence, for all crypto blobs found, we check if slot 465 is not a zero buffer. In case a crypto blob with this property is found, we have recovered all the cryptographic secrets needed for a full recovery.

With the previous version of the crypto blob at our disposal, the next step is to instantiate it (i.e. revert the crypto blob on the drive to this previous version). As discussed in Section VI-D, a vendor-specific command exists that conveniently allows us to do so. Once executed, at this point, in case the drive was protected through ATA security, the contents are accessible as normal. In the case of TCG Opal, the drive is in an in-between state, in the sense that the cryptographic secrets are known to the drive, but it still requires a password. However, this is a minor obstacle that can be overcome by, once more, modifying the password validation routine through JTAG so that it accepts any password. Finally, the drive can be unlocked as normal through `sedutil`, with any password.

### *E. Samsung 850 EVO*

The Samsung 850 EVO is a SATA SSD released in 2014. Similar to the 840 EVO, it features TLC NAND with an SLC write cache, and supports TCG Opal version 2. It is based around Samsung's MGX controller, which, contrary to the 840 EVO, is a dual-core Cortex R4.

*Firmware:* Similar to the 840 EVO, firmware updates can be downloaded through Samsung's website that come as bootable ISO images. The firmware image is once again obfuscated, though the obfuscation function is different. De-obfuscation is still performed on the host PC. The image is encrypted with AES-256 in ECB mode. The key is contained within the update utility executable, as a BASE64-encoded string. As is the case with the 840 EVO, the firmware image is cryptographically signed with ECDSA. The implementation is likely a copy of that of the 840 EVO.

*Debugging:* The 850 EVO has the exact same JTAG pin layout that the 840 EVO also has (Figure 5).

*Findings:* The motivation for analyzing the 850 EVO internals is twofold. Firstly, it is valuable to verify whether the weaknesses identified in the 840 EVO are also present in their successor. Secondly, the 850 EVO supports DEVSLP, and other drives of the same family likely use the same or a very similar implementation. In case DEVSLP is not implemented carefully, it may compromise the encryption (Section V-E).

**Key derivation scheme** The Opal key derivation scheme has not changed significantly since the 840 EVO. The implementation is still based around a crypto blob, although the number of slots has doubled, resulting in a 128 KB crypto blob. The exact reason for this remains to be researched. The Opal key derivation scheme is identical, except for a change of slot numbers. Furthermore, the vendor-unique commands listed in Section VI-D have remained unaltered.

**DEVSLP mode** In case the DEVSLP signal is received, Core 1 encrypts all secret key information present in its private SRAM region using AES-XTS. The key used is the output of PBKDF2(HMAC_SHA256, `"yoochan.kim@samsung.com"`, `"This is 4DEVSLP"`). Hence, it is constant for all drives. Once the encryption has finished, the contents of the SRAM is copied to DRAM. Four 'magic' numbers are written to DRAM, and finally, the cores and SRAM are powered down.

In order to determine whether portions of secret key information reach non-volatile storage, we reverse engineered the boot process of the drive. A diagram picturing the code flow during the boot process is given in Figure 9.

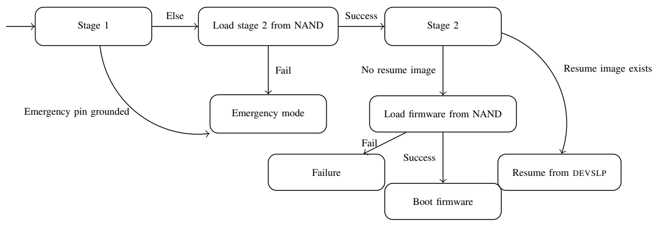

Fig. 9. Samsung 850 EVO boot process.

The first portion of code is, what we refer to as, the Stage 1 boot loader. Presumably, it is based in ROM. Essentially, its purpose is to retrieve Stage 2 from NAND and execute it. However, in case the emergency pin (Figure 5) is grounded, or in case the firmware cannot be retrieved, the drive goes into an emergency state. In this state, the drive accepts firmware images through a proprietary protocol layered over UART. The protocol was reverse engineered by [10].

Once Stage 2 is reached, the DRAM is initialized. Shortly after, the decision is made to either resume from a previous state, or to perform a normal startup procedure. The decision is made based on hardware I/O address `0x10050040`, bit 3. Before reverting to the previous state, a check is performed on whether the magic numbers written to RAM previously have remained unaltered.

No I/O addresses related to NAND are interacted with, indicating that the DRAM is kept powered during DEVSLP. We devised the following steps in order to confirm it:

(i) Modify a firmware image, such that within the Stage 2 boot loader, all references to `0x10050040` are replaced so that a DEVSLP resumption scenario is simulated. Furthermore, at the point in the code where the magic numbers are checked, an infinite loop is inserted.
(ii) Modify the currently running firmware in RAM such that it accepts firmware updates with invalid signatures.
(iii) Flash the modified firmware image through the ATA 0X92 DOWNLOAD MICROCODE command. The drive will not reboot.
(iv) Send the DEVSLP signal. The drive goes into DEVSLP mode.
(v) Power up the drive by sending the DEVSLP signal again.
(vi) The execution is stuck at the point where the infinite loop is inserted. Halt the execution and verify that the magic numbers in DRAM are present.
(vii) Power down the drive by removing the power plug.
(viii) Power it up again. The execution is stuck at the same point. In case the magic numbers still exist in DRAM, they must have originated from non-volatile storage. If absent, either the non-volatile storage device is erased during (v), or volatile storage is used.
(ix) Use the emergency mode to flash an unmodified version of the firmware, repeat all previous steps and omit (v) and (vi). Absence of the magic values in DRAM confirms that volatile storage is used.

By pursuing the above steps, we confirmed that the secret key information is indeed kept in volatile storage. The reason for encrypting it with a constant key remains unclear.

*Security evaluation:* With respect to the implementation of full-disk encryption, the 850 EVO is very similar to its predecessor. Regarding ATA security; we verified that, as with the 840 EVO, the drive can be tricked into granting access to its contents, in case the MASTER PASSWORD CAPABILITY bit is set to High.

Since TCG Opal implementation is mostly identical to its predecessor, no weaknesses have been identified. As is the case with the 840 EVO, the 850 EVO features a hardware RNG, with the added possibility to use a PRNG based on AES.

**Wear leveling** Unfortunately, despite numerous efforts, we were unable to identify the routines responsible for storing/retrieving the crypto blob from NAND flash. However, during the responsible disclosure trajectory, a contact at Samsung informed us that from the 850 EVO series onward, the crypto blob storage is no longer wear leveled. Instead, a fixed physical address in NAND is used for the crypto blob storage. Therefore, contrary to its predecessor, the 850 EVO is not vulnerable to the crypto blob recovery attack presented in the previous section.

**DEVSLP mode** We confirmed that, while the drive is in DEVSLP mode, secret key information is stored in volatile memory that is kept powered. Therefore, the DEVSLP implementation introduces no security vulnerabilities.

*Attack strategy:* The attack strategy is identical to that of the 840 EVO, with the exception of the wear leveling issue not being present. See Section VI-D.

## *F. Samsung T3 portable*

The Samsung T3 portable SSD is an external drive connected through USB-3.1 Gen 1. It offers optional password protection through a proprietary command set. The drive comes with a tool that allows the user to set or remove a password, lock and unlock.

Opening up the drive uncovers that it is essentially an 850 EVO mSATA behind a USB to mSATA bridge, albeit fitted with a special firmware supporting the proprietary command set. No firmware image for this drive can be found online. Fortunately, as is the case with the 840 EVO and 850 EVO, the drive has a JTAG interface exposed on the PCB (Figure 5). Through JTAG, the currently running firmware can be pulled from RAM.

Capturing USB packets with the help of Wireshark during locking and unlocking of the drive reveals that the ATA opcode `0x8e` (vendor-specific) is used for both operations. Analysis of the firmware reveals that the implementation of the operations is built upon the ATA security functionality of the 850 EVO. However, it resembles the behavior observed when the MASTER PASSWORD CAPABILITY bit is set to High. Thus, the password is not cryptographically linked to the DEK.

*Attack strategy:* The password validation routine can be bypassed by modifying the running firmware in RAM through JTAG in such a way that it accepts any password. We confirmed that, by doing so, the drive unlocks successfully with an arbitrary password.

### G. Samsung T5 portable

The Samsung T5 portable SSD is the successor of the T3. It uses the same MGX controller found in the 850 EVO and the T3. A notable difference between the T5 and its predecessor is that its USB to mSATA converter support for USB-3.1 Gen 2.

Another important difference is that the JTAG feature is disabled. Additionally, the emergency pin (Figure 5) is also no longer functional. Finally, no firmware updates for the T5 are available for download. Hence, for this drive, we do not have a firmware image at our disposal.

The T5 features the same vendor-specific commands found in all other Samsung SSDs (Section VI-D). Thus, despite the lack of a firmware image and debugging capabilities, the crypto blob can still be transferred from/to the device.

We retrieved a copy of the crypto blob by means of the vendor command both before and after setting a password, and inspected the differences. We refer to these blobs as $B_0$ and $B_1$, respectively. The crypto blobs are encrypted (obfuscated) with a per-device key stored within the controller itself. As such, it can only be retrieved through JTAG or unsigned code execution. Both of which we do not have. However, since XTS mode is used, we can observe whether or not the two blobs differ on a per-block (16 bytes) granularity. By studying the T3 firmware, and assuming the implementation is broadly the same, we found that the differences between $B_0$ and $B_1$ are explained by the following modifications to the plain-text crypto blob:

(i) The crypto blob revision number.
(ii) The bitmap determining for each slot in the crypto blob whether or not it is in use.
(iii) The key storage data structure for the password (Figure 6).
(iv) The so-called 'security state' byte (referred to in the firmware as such).

Given the above, by using only the vendor-specific crypto blob storage and retrieval commands, we can already confirm (or refute) that the cryptographic binding between password and DEK is absent on the T5 as well. We do so by reverting the security state byte to its previous state. In the absence of cryptographic binding, the security state byte alone likely determines the locking state of the drive, and reverting it will result in the drive being unlocked. We create a new crypto blob $B_1'$, which is constructed by taking $B_1$ and selectively reverting the 16-byte block containing the security state byte by taking its ciphertext value from $B_0$. Subsequently we upload the $B_1'$ crypto blob to the drive through the designated vendor-specific command.

We found that the drive successfully unlocks after pursuing the steps above, confirming that cryptographic binding between password and DEK is indeed absent.

*Attack strategy:* Although the steps given above confirm that the T5 lacks cryptographic binding between password and DEK, the steps themselves do not serve as an attack strategy, as (a portion of) the crypto blob from a previous state, $B_0$, is needed. However, as we confirmed, protection of the user data is not cryptography enforced. Hence, a means of low level control over the device, e.g. unsigned code execution, will allow us to bypass it.

Acquiring unsigned code execution on the device is considerably time-consuming and labor-intensive. Given that we exploited the issue in practice on the T5's predecessor, the T3, and given that the exact same issue is confirmed to exist in the T5, it is in our opinion justified to skip the act of acquiring code execution on the T5, solely for the purpose of developing an exploit for this issue.

For completeness: unsigned code execution may by accomplished via one of the methods described in Section IV-B2. Once accomplished, one can deploy the same strategy as with the T3 (Section VI-F), i.e. modifying the password routine in RAM so that it accepts any password, and subsequently unlocking the drive as normal with an arbitrary password.

## VII. Discussion

An overview of possible flaws in hardware-based full-disk encryption was given. We have analyzed the hardware full-disk encryption of several SSDs by reverse engineering their firmware, with focus on these flaws. The analysis uncovers a pattern of critical issues across vendors. For multiple models, it is possible to bypass the encryption entirely, allowing for a complete recovery of the data without any knowledge of passwords or keys. Table I gives an overview of the models studied and the flaws found.

The situation is worsened by the delegation of encryption to the drive if the drive supports TCG Opal, as done by BitLocker. In such case, BitLocker disables the software encryption, relying fully on the hardware implementation. As this is the default policy, many BitLocker users are unintentionally using hardware encryption, exposing them to the same threats.

The results presented in this paper show that one should *not rely solely on hardware encryption as offered by SSDs for confidentiality*. We recommend users that depend on hardware encryption implemented in SSDs to employ also a software full-disk encryption solution, preferably an open-source and audited one. In particular, VeraCrypt allows for in-place encryption while the operating system is running, and can coexist with hardware encryption. Furthermore, BitLocker users can change their preference to enforce software encryption even if hardware encryption is supported by adjusting the Group Policy setting[2]. However, this has no effect on already-deployed drives. Only an entirely new installation, including setting the Group Policy correctly and securely erasing the internal drive, enforces software encryption. VeraCrypt can be an alternative solution for these existing installations, as it offers in-place encryptions.

It is important to ask ourselves what problem SEDs are actually trying to address. As described in Section III, SEDs

[2]See https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-R2-and-2012/jj679890(v=ws.11)#configure-use-of-hardware-based-encryption-for-fixed-data-drives.

| Drive | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Impact |
|---|---|---|---|---|---|---|---|---|---|---|
| Crucial MX100 (all form factors) | ✗ | ✗ | ✗ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ✗ Compromised |
| Crucial MX200 (all form factors) | ✗ | ✗ | ✗ | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | ✗ Compromised |
| Crucial MX300 (all form factors) | ✓ | ✓ | ✓ | ▨ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ Compromised |
| Samsung 840 EVO (SATA) | ✗ | ✓ | ✓ | ▨ | ✓ | ✓ | ✓ | ✗ | ✓ | ∼ Depends |
| Samsung 850 EVO (SATA) | ✗ | ✓ | ✓ | ▨ | ✓ | ✓ | ✓ | ✓ | ✓ | ∼ Depends |
| Samsung T3 (USB) | ▨ | ▨ | ▨ | ✗ | ▨ | ▨ | ▨ | ▨ | ▨ | ✗ Compromised |
| Samsung T5 (USB) | ▨ | ▨ | ▨ | ✗ | ▨ | ▨ | ▨ | ▨ | ▨ | ✗ Compromised |

1 Cryptographic binding in ATA Security (High mode)
2 Cryptographic binding in ATA Security (Max mode)
3 Cryptographic binding in TCG Opal
4 Cryptographic binding in proprietary standard
5 No single key for entire disk
6 Randomized DEK on sanitize
7 Sufficient random entropy
8 No wear leveling related issues
9 No DEVSLP related issues

TABLE I
OVERVIEW OF CASE STUDY FINDINGS.

do not offer any meaningful mitigations in situations where software encryption falls short. However, as demonstrated, in situations where software encryption offers full data confidentiality, hardware encryption often does not. Hence, at best, the security guarantees of SEDs are similar to that of software encryption, and often much less. Finally, nowadays since the AES-NI extension on x86 CPUs has become mainstream, it seems legitimate to question the supposed performance and side-channel susceptibility benefits of SEDs as well.

Hardware encryption currently comes with the drawback of having to rely on proprietary, non-public, hard-to-audit crypto schemes designed by their manufacturers. Correctly implementing disk encryption is hard and the consequences of making mistakes are often catastrophic. For this reason, implementations should be audited and subject to as much public scrutiny as possible. Manufacturers that take security seriously should publish their crypto schemes and corresponding code so that security claims can be independently verified.

A pattern of critical issues across vendors indicates that the issues are not incidental but structural, and that we should critically assess whether this process of standards engineering actually benefits security, and if not, how it can be improved. The complexity of TCG Opal contributes to the difficulty of implementing the cryptography in SEDs. From a security perspective, standards should favor simplicity over a high number of features. The requirements as specified by the Opal standard, having a many-to-many relation between passwords and keys, and allowing for multiple independent ranges with adjustable bounds, makes it very hard to implement it correctly.

Finally, TCG should publish a reference implementation of Opal to aid developers. This reference implementation should also be made available for public scrutiny. It should take into account that wear-leveling is applied for non-volatile storage. Opal's compliance tests should cover the implementation of the cryptography and these tests should be independently assessed.

## REFERENCES

[1] Ieee standard for cryptographic protection of data on block-oriented storage devices. *IEEE Std 1619-2007*, pages c1–32, April 2008.

[2] Gunnar Alendal, Christian Kison, and modg. got hw crypto? on the (in) security of a self-encrypting drive series. *IACR Cryptology ePrint Archive*, 2015:1002, 2015.

[3] David Clunie, Rich Shroeppel, Phillip Rogaway, Vijay Bharadwaj, and Neils Ferguson. Public comments on the xts-aes mode. *Collected email comments released by NIST, available from their web page*, 2008.

[4] J Domburg. Hard disk hacking, 2013. See http://spritesmods.com/?art=hddhack.

[5] J Domburg and Tweakers.net. Secustick gives false sense of security, 2007. See https://tweakers.net/reviews/683/secustick-gives-false-sense-of-security.html.

[6] Niels Ferguson. Aes-cbc+ elephant diffuser: A disk encryption algorithm for windows vista, 2006.

[7] Travis Goodspeed. Active disk antiforensics and hard disk backdoors. In *Talk at 0x07 Sec-T Conference (video: https://www.youtube.com/watch?v=8Zpb34Qf0NY)*, volume 8, 2014.

[8] J Grand. Jtagulator: assisted discovery of on-chip debug interfaces. In *21st DefCon Conference, Las Vegas*, 2013.

[9] Trusted Computing Group. Tcg storage security subsystem class: Opal specification version 2.01, 2015.

[10] P Gühring. The missing Samsung EVO 840 - 250 GB SSD repair manual. See http://www2.futureware.at/~philipp/ssd/TheMissingManual.pdf, 2016-2018.

[11] J Horchert, J Appelbaum, and C Stöocker. Shopping for spy gear: Catalog advertises nsa toolbox. *Der Spiegel*, 2013.

[12] Tilo Müller, Felix C Freiling, and Andreas Dewald. Tresor runs encryption securely outside ram. In *USENIX Security Symposium*, volume 17, 2011.

[13] Tilo Müller, Tobias Latzo, and Felix C Freiling. Self-encrypting disks pose self-decrypting risks. In *the 29th Chaos Communinication Congress*, pages 1–10, 2012.

[14] Tilo Müller, Benjamin Taubmann, and Felix C Freiling. Trevisor. In *International Conference on Applied Cryptography and Network Security*, pages 66–83. Springer, 2012.

[15] T Ptacek and E Ptacek. You don't want xts, 2014. See https://sockpuppet.org/blog/2014/04/30/you-dont-want-xts/.

[16] Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes ocb and pmac. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 16–31. Springer, 2004.

[17] CE Stevens. AT Attachment 8-ATA/ATAPI Command Set – 4 (ACS-4). *Working Draft, American National Standard, Revision 14*, 2016.

[18] Roel Verdult. *The (in) security of proprietary cryptography*. Sl: sn, 2015.

[19] J Wetzels and A Abbasi. Wheel of fortune: Analyzing embedded os random number generators. 2016.

[20] Jonas Zaddach, Anil Kurmus, Davide Balzarotti, Erik-Oliver Blass, Aurélien Francillon, Travis Goodspeed, Moitrayee Gupta, and Ioannis Koltsidas. Implementation and implications of a stealth hard-drive backdoor. In *Proceedings of the 29th annual computer security applications conference*, pages 279–288. ACM, 2013.

## Appendix A
## Execution trace captured on a Crucial MX300 drive during the BitLocker set-up phase.

```
VerifyPasswd(szPasswd="AEGIS_ACADIA_MSID_12456789012345", bExtractRdsKey=true, dwSlotNo=2)
VerifyPasswd(szPasswd="AEGIS_ACADIA_MSID_12456789012345", bExtractRdsKey=true, dwSlotNo=2)
CopyCredential(dwSourceSlot=2, dwDestinationSlot=10)
ProtectPasswd(szPasswd=[0x00 × 32], bStoreRdsKey=true, dwSlotNo=11)                    ▷ szPasswd is zero buffer
CopyCredential(dwSourceSlot=11, dwDestinationSlot=12)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=13)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=14)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=15)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=16)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=17)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=18)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=19)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=20)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=21)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=22)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=23)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=24)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=25)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=26)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=27)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=28)
CopyCredential(dwSourceSlot=11, dwDestinationSlot=29)
StoreCryptoContextInSpiFlash()
VerifyPasswd(szPasswd="AEGIS_ACADIA_MSID_12456789012345", bExtractRdsKey=true, dwSlotNo=2)
VerifyPasswd(szPasswd="AEGIS_ACADIA_MSID_12456789012345", bExtractRdsKey=true, dwSlotNo=10)
VerifyPasswd(szPasswd="AEGIS_ACADIA_MSID_12456789012345", bExtractRdsKey=true, dwSlotNo=2)
ProtectPasswd(szPasswd=«BitLocker SID password», bStoreRdsKey=true, dwSlotNo=2))
StoreCryptoContextInSpiFlash()
VerifyPasswd(szPasswd="AEGIS_ACADIA_MSID_12456789012345", bExtractRdsKey=true, dwSlotNo=10)
ProtectPasswd(szPasswd=«BitLocker SID password», bStoreRdsKey=true, dwSlotNo=10)
StoreCryptoContextInSpiFlash()
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
GenerateRandomDekAndWrap(dwRangeNo=1, bIsProtectedRange=false)
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
StoreCryptoContextInSpiFlash()
UnwrapDek(dwRangeNo=1, bIsProtectedRange=false)
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
UnwrapDek(dwRangeNo=1, bIsProtectedRange=false)
WrapDek(dwRangeNo=1, bIsProtectedRange=true)
VerifyPasswd(szPasswd=[0x00 × 32], bExtractRdsKey=true, dwSlotNo=15)
ProtectPasswd(szPasswd=«BitLocker user password», bStoreRdsKey=true, dwSlotNo=15)
StoreCryptoContextInSpiFlash()
VerifyPasswd(szPasswd=«BitLocker user password», bExtractRdsKey=true, dwSlotNo=15)
VerifyPasswd(szPasswd=«BitLocker user password», bExtractRdsKey=true, dwSlotNo=15)
```

**Algorithm 3** UnwrapDek

```
Require: abRdsKey, abDeviceKey, aabRangeKeyTable,
         aabUnwrappedRangeKeyTable
Ensure: Range key dwRangeNo is unwrapped
  function UNWRAPDEK(dwRangeNo, bIsProtectedRange)
      if bIsProtectedRange then
          abKey ← abRdsKey
      else
          abKey ← abDeviceKey
      abCiphertext ← aabRangeKeyTable[dwSlotNo]
      abPlaintext ← DECRYPT(abKey, abCiphertext)
      if decrypt failed then
          return ERROR
      aabUnwrappedRangeKeyTable[dwSlotNo] ← abPlaintext
      return SUCCESS
```

## Appendix B
## Pseudocode of various routines in the Crucial MX300 firmware.

The ProtectPasswd function (Algorithm 1) takes a password and stores it in the credential table so that an incoming password can be checked for validity at a later point in time. The bStoreRdsKey parameter determines whether the stored credential should encapsulate the RDS key (discussed in the previous section). In this case, the credential allows access to protected ranges (see Figure 4).

**Algorithm 1** ProtectPasswd

```
Require: abRdsKey, abDeviceKey, aabCredentialTable
Ensure: Credential szPasswd is stored in aabCredentialTable at dwSlotNo
  procedure PROTECTPASSWD(szPasswd, bStoreRdsKey, dwSlotNo)
      if bStoreRdsKey then
          abPlaintext ← abRdsKey
      else
          abPlaintext ← [0x00 × 32]            ▷ abPlaintext is a zero buffer
      abSalt ← RANDOM(32 bytes)
      abKey ← PBKDF2(szPasswd, abSalt)
      abCiphertext ← ENCRYPT(abKey, abPlaintext)
      stProtectedPasswd ← (abSalt, abCiphertext)
      abOutput ← ENCRYPT(abDeviceKey, stProtectedPasswd)
      aabCredentialTable[dwSlotNo] ← abOutput
```

The function VerifyPasswd (Algorithm 2) is the inverse of ProtectPasswd. It has two purposes: checking the validity of a password, and, in case the bExtractRdsKey parameter is set, using the password to decrypt the RDS key and copying it to the global RDS key buffer, allowing other functions to use it.

**Algorithm 2** VerifyPasswd

```
Require: abRdsKey, abDeviceKey, aabCredentialTable
Ensure: Verify szPasswd and set global RDS key if bExtractRdsKey = true
  function VERIFYPASSWD(szPasswd, bExtractRdsKey, dwSlotNo)
      abInput ← aabCredentialTable[dwSlotNo]
      stProtectedPasswd ← DECRYPT(abDeviceKey, abInput)
      if decrypt failed then
          return ERROR
      (abSalt, abCiphertext) ← stProtectedPasswd
      abKey ← PBKDF2(szPasswd, abSalt)
      abPlaintext ← DECRYPT(abKey, abCiphertext)
      if decrypt failed then
          return ERROR
      if bExtractRdsKey then
          abRdsKey ← abPlaintext
      return SUCCESS
```

Furthermore, the UnwrapDek function (Algorithm 3) takes an entry from the range key table, and decrypts it using either the RDS key, or the device key (for protected and unprotected ranges, respectively), as determined by the bIsProtectedRange parameter. Obviously, for protected ranges, the RDS key must be decrypted, prior to invoking UnwrapDek.

Finally, we implicitly define the functions WrapDek, Copy-Credential, GenerateRandomDekAndWrap, and StoreCrypto-ContextInSpiFlash as their functionality is clear from their names.