Towards Trustable Storage Using SSDs with Proprietary FTL

Xiaotong Cui, Minhui Zou, Liang Shi and Kaijie Wu[§]
Key Laboratory of Dependable Service Computing in Cyber Physical Society, Ministry of Education,
College of Computer Science, Chongqing University, Chongqing, China. 400044
Email: {xiaotong.sd, shi.liang.hk, kaijie}@gmail.com, zouminhui@outlook.com

***Abstract*—In recent years we have seen an increasing deployment of flash-based storage, such as SSD, in mission-critical applications due to its fast read/write speed, small form factor, strong shock resistance, and etc. SSDs use a middle layer called flash translation layer (FTL) to maintain the compatibility with the traditional magnetic-based HDDs. Unlike the traditional HDD where the host OS has the knowledge on where and how to access data, SSD uses FTL to translate and implement all operations. Even worse, FTL, which is considered as one of most important intellectual properties of flash-based storage, is often proprietary. This brings up a serious security concern on design trustiness: what if the manufacturer either accidentally or intentionally implements those operations incorrectly or maliciously? In this paper we analyze the possible threats that are brought up by the design trust issues, and propose a simple yet effective countermeasure.**

I. Introduction

Flash-based storage has played an important role in bridging up the gap between the processing speeds of CPU and the I/O speeds of secondary storage. In recent years, we have seen an strong trend on replacing the traditional magnetic-based Hard Disk Drives (HDD) to flash-based storage such as Solid State Drives (SSD). Compared with HDD, SSD has less random read/write latency, less power consumption, better shock resistance, and etc [1]. However, the data organization of SSD is very different from the HDD.

A HDD comprises one or more platters, each platter has many co-centric tracks, and each track has many sectors. The tracks with the same radius vertically across all platters construct a cylinder. Each platter is accessed by two headers. Hence a Cylinder-Head-Sector (CHS) tuple uniquely addresses the smallest unit of data storage, as shown in Fig. 1. The Logical Block Address (LBA), which is used by the host OS to address the data of files, is simply a linear function of a CHS tuple. Hence, the host OS indirectly has a full control over all physical blocks where data are stored.

SSD, on the other hand, uses flash technology – a completely different mechanism from HDD. The organization of data is also different, as shown in Fig. 2. A SSD usually consists of a number of channels, where each channel connects to a handful NAND flash chips. Each chip has multiple dies, each die has multiple planes (typically 2 or 4), each plane has multiple blocks, and each block has multiple pages. A page usually has 4KB or more. Hence in this architecture, a LBA, which is a linear mapping from a CHS tuple in

[§] This work is supported by the Fundamental Research Funds for the Central Universities (Project No.106112014CDJZR185502). Kaijie Wu is the corresponding author: kaijie@gmail.com.

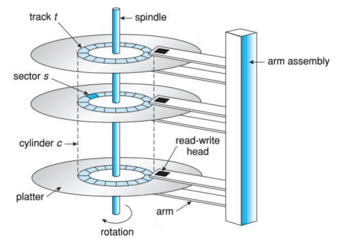


Fig. 1. The CHS addressing of HDD [2].

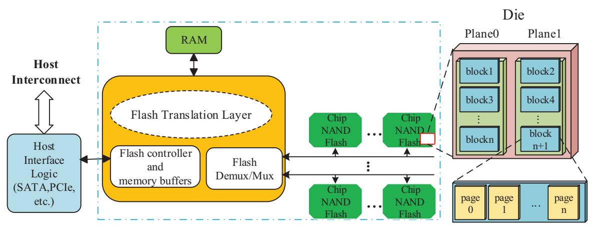


Fig. 2. The structure of a SSD connecting with a HOST

HDD, has no direct mapping to a physical page in SSD. To maintain the compatibility, SSD uses a middle layer called Flash Translation Layer (FTL) to translate and implement the operations originated from the host OS. As a firmware of flash controller, FTL translates logical block addresses, which is used by the host OS for data location, to raw flash addresses that identify the physical pages in the SSD. It hides the inner characteristics of SSD organization and exposes only an array of logical blocks to the host such that data can be read or written with a fixed protocol regardless of the detailed specifications (such as page size, block size, SLC or MLC technology, etc). In addition to serving the read/write/delete requests from host, the functionalities of FTL also includes garbage collection, wear-leveling and bad block management in the background which are transparent to the host and important as well for the reliability of the system [3].

As the core "intelligence" of the controller, FTL is considered as one of most important intellectual property of SSD design, which is often proprietary to the manufacturer [4]. Despite the fact that we have spent tremendous efforts to achieve a trustable design and implemented a whole package of methods such as maintaining the root of trust, ensuring a chain-of-trust, and etc, the proprietary FTL solely controlled by the manufacturer leaves a big hole of trust: what if the manufacturer either accidentally or maliciously implements those tasks incorrectly?

If the incorrection is implemented accidentally (i.e., a software bug), it could affect the system dependability, but can be corrected by software update. However, if the incorrection is implemented maliciously, the situation becomes much worse, and could be out of control if no proper action is followed. A maliciously implemented FTL can perform tasks correctly for the most of time, but will become "evil" upon a trigger. The part of evil-behave code is considered as the "Manufacturer-Injected Trojan", and can be hided easily in the program due to the proprietary feature of FTL. It is more stealthy than the Trojans injected by external attackers, and hence poses more serious threats. Manufacturer-Injected Trojan in IC design and manufacturing has attracted a lot of attention in recent years [5] [6] and has become a hot topic in major conferences such as DAC [7] and ASP-DAC [8]. Trojan circuits in ICs could be introduced in any step of the design and manufacturing process as long as the step is not carried by a trustable vendor. We believe the same trust issue applies to the FTL as well due to its proprietary feature. In this work, we will analyze the possible threats brought up by the trust concern and propose a simple yet effective approach to counter the threats.

The rest of paper is organized as follows, Section II motivates the work and introduces the threat models, Section III analyzes the threat models with possible solutions, Section IV proposes a comprehensive countermeasure, and Section V is conclusion.

II. Motivation and The Threat Model

This paper is dedicated to the potential threats posed by a maliciously implemented FTL. While the proprietary feature helps protect the business benefits of the vendor, it poses a serious design trust issue: what if the FTL is maliciously implemented with evil-behave "Trojan" function? The Trojan function can remain inactive for most of time, and can be active upon a trigger. Unlike the external-injected Trojans, manufacturer-injected Trojan in a proprietary program such as FTL is easy to hide, and can make any level of damage as it wish! This poses a serious threat to the trustable design built on top of it!

Such malicious trojan usually comprises two parts, one is trigger and the other is payload. Both trigger and payload can be implemented as a piece of evil-behave code, or as a piece of small circuits hardwired in the flash controller. There are many reports on how to do this [5] [9]. Once the Trojan is triggered, the Payload part will be active to cause damage to the system. Potential threats can be classified into 5 aspects according to the outcome of the Payload.

Threat 1. Loss of Data: The Payload of Trojan can either delete data by erasing selected or all blocks, or hijack data by encrypting data and ask for price.

Threat 2. Lifetime Reduction: As is known to all, flash memory cells have limited numbers of program/erase cycles, which makes Wear-Leveling an important function of FTL. Thus tuning done the effectiveness of the Wear-Leveling function or completely disabling it could be one potential threat. As a result, *Threat 2* can cause a (significant) reduction in the lifetime of SSD.

Threat 3. Performance Reduction: It is easy for the maliciously implemented FTL and the flash controller to slow down the overall performance by performing unnecessary operations, which results in the delay/extension of response time to the real requests from host.

Threat 4. Data Integrity Damage: The FTL maintains the LBA-PBA table that maps logical address to physical address. The Payload can easily change the original LBA-PBA table and leads the mapping to harmful data or executable code. As the result, the integrity of the stored data could be damaged.

Threat 5. Data Stealing: Stealing the data stored in the SSD is an easy job for the Payload. There are couple ways of doing so in a SSD. The first way is via the "bad" blocks since they are transparent to the host. Another way is to use the stealth capacity to store the stolen data. A manufacturer can build a drive with 200 GB capacity but only report 100 GB to the host. And upon the drive disposal, data can be read back easily by hackers. Even the latest data sanitization technique specifically designed for SSDs fails to solve this problem [10]!

As can be seen above, with a maliciously implemented FTL, there exist many security threats and all of them can be implemented with trivial efforts. This leaves a significant security hole to the trustable design built on top of it. In the following sections, we will analyze these threats one by one and propose a comprehensive countermeasure.

III. Analysis of Threat Model

Many of the threats outlined in Section II effectively enforce a denial of service attack to the system that relies on the SSD. To counter these threats, mirroring technique such as RAID-1 is promising – if one fails, the other one can continue the service until the failed one is replaced. The RAID technologies, however, were developed for storage dependability, but not for security threats. If the redundant drives are from the same manufacturer, they will fail the same way! In this work, as a baseline countermeasure, we propose a RAID-1 like scheme with an important additional requirement: the redundant drives must come from two different manufacturers. It is unlikely that both drives are maliciously implemented, hence the redundancy can be used to counter the threats. In the worst case where both drives are maliciously implemented, it is extremely unlikely that both drives will be triggered simultaneously and their payloads work the same way. In the following, we will walk through the threats and analyze the effectiveness of the baseline scheme, and will propose improvement if the baseline scheme does not counter the threats.

Analysis of *Threat 1*. Recall that *Threat 1* is the loss of data due to maliciously data deletion, encryption, and etc. Apparently the baseline scheme, the RAID-1 like scheme with the diverse manufacturer requirement, counters this threat. Write requests will be performed on both drives and read requests can be serviced by any of drives. Once data in one SSD is lost, data in the other can be used.

Analysis of *Threat 2*. Recall that *Threat 2* is the lifetime reduction caused by disabling the wear-leveling or tuning down its effectiveness. Apparently the baseline scheme counters this threat. Just replacing the failed SSD when it reaches the end of its lifespan ensures the continuous service.

Analysis of *Threat 3*. Recall that *Threat 3* is performance reduction due to unnecessary operations. Again, the baseline scheme counters this threat. The system can continue forward as long as one drive finishes the requests.

Analysis of *Threat 4*. Recall that *Threat 4* is the damage on data integrity as the data stored in SSD could be replaced with malicious information. In this case, the baseline scheme fails to counter this threat. This is because while the baseline scheme provides both valid and invalid data upon read requests, there is no way to differentiate which one is valid. It is reasonable to assume that when the Payload of Trojan replaces the valid data with malicious data, it will make the malicious data look like "valid" by providing correct ECC codings. Otherwise, its attempt will be detected simply by checking the correctness of the data. As a result, extra functionality is needed for this purpose.

One straightforward solution is to provide triple redundancy, i.e., 3 drives, and all from different manufacturers. But the cost is high. In this work we propose another method that only requires 0.4% extra capacity, as an improvement to the baseline scheme. The improvement is to maintain a record of digests of all the pages that are written to the SSD drives at the host side. Upon a write request of a logical page, the host will update its corresponding digest in the record at the same time when the request is serviced. Upon a read request of a logical page, the host will check the integrity of the page returned by a drive using the record, with a miss match indicating an error! Hence if the data integrity of one drive is damaged, the damage can be detected and the data will be discarded without interrupting the service. Frequently failing on integrity is a sign of malicious implementation.

Analysis of *Threat 5*. Recall that *Threat 5* is data stealing using non-claimed storage space. The baseline scheme, even equipped with the improvement, does not counter the threat, neither does the latest data sanitization technique developed specifically for SSDs [10]. In this case we suggest, upon a drive disposal, either continue keep it securely until the value of data expires, or physically destroy the chips.

IV. The Comprehensive Countermeasure

According to the analysis given in Section III, the comprehensive countermeasure shall have the following features to counter all identified threats:

- It must use a mirrored architecture with redundancy, and the redundant drives must be from independent manufacturers (shown in Fig. 3) – This feature counters the loss of data, shortened lifespan, and reduced performance caused by a maliciously implemented FTL.
- The host needs to maintain a record of digests for all pages that are written to drives – This feature counters the integrity damage caused by a maliciously implemented FTL.

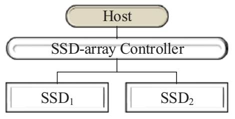


Fig. 3. Mirrored architecture. Note, SSD1 and SSD2 are from different manufacturers.

A SSD-array controller is needed to connect the mirrored architecture to the host. Except for implementing all the functions that a regular RAID controller will provide, it also implements some additional features to help minimize the interruptions to the host. The core part is to calculate the digests of pages for integrity checking and only pass the pages that have sound integrity to the host.

A. Implement the Record of Digests

Different hash functions such as MD5, SHAs and etc, can be used to calculate the digests of pages, which can generate different storage and performance overheads. The detailed analysis will be given later. The calculation of the digests is assigned to the host side due to trustiness problem though most of modern SSDs are equipped with powerful processor(s).

The record of digests is indexed by the logical page numbers used by the host (as shown in Table I), which is a convenient choice, since the record of digests is maintained at the host side and the host can maintain the record without involving the attached drives, and only pays attention to the pages that have passed through its own I/O.

TABLE I. Data to Message-Digest Mapping Table

MD	S_{LPN0}	S_{LPN1}	S_{LPN2}	$\ldots$	S_{LPNn}

B. The Storage of the Record of Digests

There are two important issues regarding the storage of the record of digests. One is the storage when the system is alive, the other one is the storage when the system is turned off.

1) The storage when the system is alive: – There are two possible locations to store the record of digests. The first one is the main memory of the host while the other one is the SSD-array controller.

Storing the record at the controller incurs minimal involvement from the host. While the downside of this approach is that it requires dedicated memory space at controller, which is not negligible according to the overhead analysis given below.

Storing the record in the main memory requires the host to maintain it: the digest calculating and record matching/updating are now performed by the host CPU. The memory space of the record is managed by the host OS with minimal extra overhead. However, the controller has to return both pages to the host when two pages from two drives don't match, which puts the host on a risk by sending a contaminated page to the system.

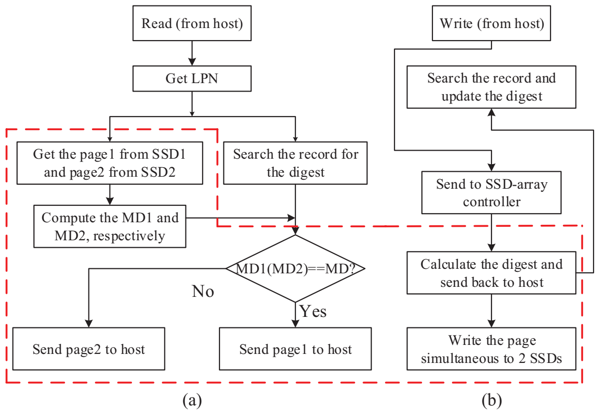
In order to optimize the trade-offs between memory space, calculation, and I/O traffic, we propose to separate the storage of the record of digests and digest calculating to the host and SSD-array controller, respectively. Upon issuing a read request of a page, the host will also pass the digest of this page to the controller. When the controller receives the pages returned from the drives, it calculates the digests and matches it to the one passed from host. Upon a write request, the controller calculates the digest and returns it back to the host. The host updates the record. By doing this way, we leave the record stay in the main memory so host OS can do its housekeeping with minimal extra overhead, and let digest calculation to be done at controller to avoid the extra I/O traffic and unnecessary computing. Digest computing using cryptographic hash algorithm such as MD5 or SHA is considered as light computation, and is well suited for the embedded processors commonly seen in controllers.

2) The storage when the system is off: – When the system is off, the record must be stored on a non-volatile storage. Using another independent drive could be too costly and again invite additional trust concerns. We propose to write back the record to the two SSDs in the mirrored architecture, with one copy for each drive, and upon the system waking up, a correct record is loaded into the host.

However, in order to distinguish the correct record upon an alteration of the records by a maliciously implemented FTL, the records stored in both drives must be authenticated. The authentication can be either offered by a dedicated authentication algorithm such as HMAC [11], or by an encryption algorithm such as AES [12]. Since a block cipher does not prevent the integrity by itself, an identifier field is appended to the record before encryption. Failing to match the pre-specified identifier in a decrypted record indicates that it has been altered [11]. The Trusted Platform Module (TPM), as the foundation component to a trustable design [13], can provide the authentication/verification, or encryption/decryption function, as well as a secure key storage. To avoid replay attack, it is mandatory to update the key after every transaction, or append *nonce* such as time-stamp or sequence number to the record.

C. High-level Read/Write Process

When the host sends a write request, the message digest is updated and the data are written to the two SSDs simultaneously through the SSD-array controller. When the host sends a read request, both SSDs will work on the request and send their own copy to the controller at which their pages will be integrity-checked and then be passed to the host. The following flow diagram (Fig. 4) shows processes of read/write request.



Read (from host)
Get LPN
Get the page1 from SSD1 and page2 from SSD2
Search the record for the digest
Compute the MD1 and MD2, respectively
MD1(MD2)==MD?
No
Yes
Send page2 to host
Send page1 to host
(a)
Write (from host)
Search the record and update the digest
Send to SSD-array controller
Calculate the digest and send back to host
Write the page simultaneous to 2 SSDs
(b)

Fig. 4. Flow diagram of proposed technique. (a) Read process. (b) Write process. Note, functions in red dashed area are processed by SSD-array controller.

D. Overhead Analysis

1) Storage Overhead: Different Hash algorithms generate different sized digests, and cause different overheads. Assume that each page is $4KB$, and $MD5$ is used. A drive with 1TB has $0.25G$ pages, and each page has a 16-Byte digest. This gives total 4GB record, hence 0.4% space overhead. If we update to a more advanced Hash and double the digest size to 32 bytes for each page, the overhead simply rises to 0.8%, still less than 1%.

2) Performance Overhead: The performance overheads are incurred in boot, read, and write processes due to the added functions, such as calculating digest, calculating the authentication and then verifying, encrypting and decrypting, searching a digest in the record, and etc.

Reboot. When the system is shutting down, the overhead is incurred when the record is processed by MAC or encryption and then written to attached drives. When the system is booting, the overhead is incurred when the record is loaded into the main memory and then authenticated. It will increase the Reboot time but will not affect the run-time performance.

Read. When the host reads a page, two SSDs simultaneously serve the request. The major overhead is incurred when the controller calculates the digests of the pages returned from drives, and matches them with the digest received from the host. The overhead can be reduced if the controller returns a page as soon as one of pages successfully passes the integrity checking. Consider throughput of Digest calculation such as MD5 can be as high as 32 Gbps [14] while the current SATA 3.0 has 6 Gbps throughput [15], the performance overhead is less than 20%.

Write. When the host writes a page, the controller can pass the page down to drives immediately. Digest calculation can be performed at the same time when drives physically write the page. Hence the performance overhead is negligible.

V. Conclusion

The proprietary FTL of SSD manufacturers raises serious security concerns due to its direct control over flash chips and operations without the notice of host. In this paper we have identified, from the aspect of trustable designs, 5 potential threats that are caused by the proprietary feature of FTL. We then have proposed a mirrored architecture to counter the first 4 threats. The proposed mirrored architecture uses redundant drives from independent manufacturers, and hosts a record of digests for all the data pages. Implementation and operation of the proposed architecture are discussed, in which space overhead is less than 1% compared to RAID-1, and read performance overhead is less than 20%, write performance overhead is negligible.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance." in *USENIX Annual Technical Conference*, 2008, pp. 57–70.

[2] "The addressing method of hdd," Website, http://ntfs.com/hard-disk-basics.htm.

[3] M. Iaculo, F. Falanga, and O. Vitale, "Introduction to ssd," in *Memory Mass Storage*. Springer, 2011, pp. 213–236.

[4] "Understanding the flash translation layer(ftl) specification," Website, http://www.jbosn.com/download_documents/FTL_INTEL.pdf.

[5] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," 2009.

[6] A. Baumgarten, M. Steffen, M. Clausman, and J. Zambreno, "A case study in hardware trojan design and implementation," *International Journal of Information Security*, vol. 10, no. 1, pp. 1–14, 2011.

[7] "Design automation conference," Website, http://dac.com.

[8] "Asia and south pacific design automation conference," Website, http://www.aspdac.com/.

[9] A. Waksman, J. Rajendran, M. Suozzo, and S. Sethumadhavan, "A red team/blue team assessment of functional analysis methods for malicious circuit identification," in *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*. ACM, 2014, pp. 1–4.

[10] M. Y. C. Wei, L. M. Grupp, F. E. Spada, and S. Swanson, "Reliably erasing data from flash-based solid state drives." in *FAST*, vol. 11, 2011, pp. 8–8.

[11] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 2010.

[12] J. Daemen and V. Rijmen, *The design of Rijndael: AES-the advanced encryption standard*. Springer, 2002.

[13] S. Bajikar, "Trusted platform module (tpm) based security on notebook pcs-white paper," *White Paper, Mobile Platforms Group–Intel Corporation*, vol. 20, 2002.

[14] Y. Wang, Q. Zhao, L. Jiang, and Y. Shao, "Ultra high throughput implementations for md5 hash algorithm on fpga," in *High Performance Computing and Applications*. Springer, 2010, pp. 433–441.

[15] A. Serial, "International organization: Serial ata revision 3.0," *Gold Revision, June*, vol. 2, 2009.