Secure Deletion: Structural Artifacts in Full Disk Encryption Schemes

Andrew Brusso

April 2019

Introduction

When you have a need to handle sensitive data, two very large and important problems show up: how do I ensure that my sensitive data is stored in a format that cannot be read by an adversarial party, and how do I ensure that when I no longer need sensitive data that it can be removed or deleted in a manner that cannot be recovered by an adversarial party? Both of these problems are similar in that they involve securing private data, meaning they involve confidentiality of data, but the approaches for both are traditionally quite different.

The conventional method of securing sensitive data from being read is to use encryption, which is a method of obscuring the data in a manner that an attacker is unable to easily revert without special knowledge (e.g. a symmetric key). In most cases, the encryption used involves block ciphers such as AES or Blowfish, because they are relatively quick, simple and can be performed on arbitrary length data easily. There are a lot of options on how this encryption works, which can meet the needs of a variety of different use cases. For individuals who only need encryption for a few files, you can use file encryption solutions at the application level to encrypt just those individual files, or if you need an entire laptop secured you can install a Full Disk Encryption (FDE) scheme at the block device layer, which will allow for everything on your hard drive to be encrypted when it is read or written. There are even hardware manufacturers now implementing Full Disk Encryption at the hardware level, so that the entirety of the drive is encrypted without involving software at all.

The conventional method of removing or deleting sensitive data normally uses some variant of repeatedly overwriting the data to try to ensure that not only is the data not readable by software, but cannot be retrieved through more physical means (e.g, taking magnetic readings on a conventional hard disk drive). An optimization that can be made to this approach is to instead encrypt the data, and then rather than overwriting the data you overwrite the keys for the data, which means you trade off more expensive deletions for more expensive reads (better for Flash Memory).

Conventional deletion methods suffer from a potential problem however: although they handle sanitizing the data itself, they don't provide guarantees about the surrounding data, or potential copies of the data that might exist elsewhere on a storage device. Some solutions exist that either take care of these structural artifacts, or deliberately do not create them, but these solutions exist below the application layer, and therefore require specific hardware to work.

My research project focused on a fairly simple question: do these structural artifacts exist when using FDE schemes? That is, if you install an FDE scheme on a device, in conjunction with secure deletion, do you get rid of the structural artifacts for free? If this is the case, then you could ignore the other solutions on the market as a consumer, purchase the hardware that you want, and simply install FDE and a traditional secure deletion solution. FDE will likely be slower than a hardware solution, but if you already want to have FDE anyway, there isn't any additional cost.

To this end, I developed some custom flash firmware on a development board to emulate a Mass Storage Device in Windows, and then performed experiments by adding and removing data from the Mass Storage Device to observe how the underlying raw flash changed. This allowed me to look for signs of structural artifacts in the FDE on the raw flash that could be potentially be used to detect deletions. I additionally

developed a set of scripts for experimentation and analysis, so that the experiments I performed can easily be repeated and analyzed.

In this report, I will first discuss related work on the subject of secure deletion and Full Disk Encryption. Next, I will discuss the adversarial model that my experiments intend to emulate. I will then describe the approach taken in implementing the solution for my project, and an evaluation of my experimental setup and results. I will end the report with some concluding remarks.

Background and Related work

There is not currently a significant body of research on the subject of using Full Disk Encryption as an approach to performing Full Disk Encryption. I was able to find some instances in online forums of people suggesting its usage would work for the purpose, but there doesn't appear to be any real research on the subject. Part of this is because prior to the advent of flash memory there was not an easy way to view the encrypted content of a full disk encryption system, because it happens at the block device layer, which means the majority of its operation is entirely opaque to the user. Another reason is that these are both two fairly disjoint areas of security, and there are not a lot of people with knowledge from both subjects. Because of these reasons, most of the related work I came across was either in the space of performing Full Disk Encryption, or in the separate space of performing Secure Deletion (on plaintext data). My research, in a sense, is trying to see if a gap can be bridged between these two research spaces, and so the related work is a combination from both of these areas.

In the space of Secure Deletion, for the past 30 years DoD 5220.22-M [9] has existed detailing methodologies for meeting data erasures requirements for the Department of Defense, and for conventional hard disk drives there hasn't really been any major departures from these methods. Since these standards were drafted, the conventional approach to secure deletion has been using multiple passes of data overwrites on the hard disk until you have ensured that there are no longer any electromagnetic remnants of the data. This approach was acceptable for hard disk drives, but due to the unique constraints of flash memory, constant overwrites at the application level would not be an effective approach to remove the data, because it would significantly impact the lifespan of the device. Boneh and Lipton [1] provided the ground work for a potential solution to this, by proposing usage of encryption as an alternative to deletion. Lee et al.[7] provided one of the first solutions to this problem focused on flash memory, which used this encryption technique in a YAFFS flash memory file system as a way of shifting the problem space from deleting the data directly to instead encrypting the data, and then only having to worry about erasing the keys. In flash memory, this approach is quite effective, because the deletions won't require constant overwrites of multiple blocks, which will quickly cause the device to degrade. The problem of deletion still occurs with the keys though, where overwriting is ultimately the only guaranteed way to remove physical remnants. Reardon et al.[8] made incremental improvements on this system for flash memory secure deletion, but these solutions focused on the data being deleted itself.

A more general problem in secure deletion for Flash Memory exists: although secure deleting the data itself isn't impossible (but does require hardware specific implementation), because flash memory uses log-structured writing, deletion isn't enough to guarantee all artifacts of the data are removed. Several solutions [2][6] appeared to address removing or not creating structural artifacts, but these schemes did not address the special nature of flash memory. It wasn't until Chen et al. [3] that solution to the problem of secure deleting with respect to structural artifacts existed that was conscious of the special nature of flash memory, although minimal leakage occurs in this solution as well. An important definition comes from this paper regarding structural artifacts, which is the **Truly Secure Deletion** guarantee. In addition to sanitising the data, truly secure deletion must guarantee that nothing can be inferred about the data from the context left over after its deletion. If you were to compare two sequences of operations performed within a truly secure deletion scheme, one where some set of insertions and corresponding deletions occur, and one where neither the insertions or corresponding deletions occurred, it is impossible to discern which sequence is which [3].

In the space of Full Disk Encryption, the market is very much private and consumer-centric, so specific details on approaches to Full Disk Encryption are not as publicly available. Most schemes rely on some type of Block Cipher to implement their encryption, and a pre-boot sequence for entering a password that is used to generate a symmetric key for the the entire hard drive. I looked at several white papers and public facing

pages with respect to secure deletion, such as for Veracrypt [10], Bitlocker [4], and Symantec [5]. These formed the basis for my approach in my experiment, and guided my thoughts on how Full Disk Encryption might impact structural artificats in secure deletion.

Adverserial Model

In the FDE secure deletion scenario, the adverserial model is in some ways inherent in the problem itself. If you imagine you are an adversary trying to attack this problem, you find you are quite restricted in what you can actually do. An adversary that already has the encryption keys to an FDE device isn't incredibly interesting, since such an adversary is essentially the same as the case of plaintext data, where structural artifacts will be apparent upon close inspection of the device. When the attacker doesn't have encryption keys however, they don't really have a lot at their disposal. Even if they have a full snapshot of all the data, it will be entirely encrypted, so at first blush it will just be random data. It is only when an adversary has multiple snapshots that they can begin to observe how the data on a device is changing, and potentially make useful observations that can help determine signs of deletions occurring. The model that an attacker will have is therefore one where they can take multiple snapshots of the device, with operations occurring between snapshots. Even with multiple snapshots there are still challenges, because in order for an attacker to identify a deletion with certainty, they have to be able to identify that an insertion had occurred with certainty first. This means in order for an attacker to make a determination that a deletion occurred, they must have a minimum of 3 snapshots: one before the data was added, one after the data was added, and one after the data was deleted.

Approach

To approach this problem, there were a few components that I knew was going to be necessary, and a few I realized would be helpful along the way. I knew the most crucial component of the problem was going to require me to create a mechanism for taking snapshots of the state of a disk drive during operation, in a reliable and deliberate manner. These snapshots have to be taken at the Flash Translation Layer (FTL), because otherwise we'll be looking at plaintext data, which our adversary does not have access to. Once I was able to take snapshots, I knew I would need to construct a series of experiments involving performing a specific set of actions, with the purpose of observing changes in the state of the disk drive, and then determine a way of easily analyzing how the FDE scheme behaved in response to these actions.

The problem that definitely proved the most difficult was simply getting a snapshot from my development board reliably, accurately, and quickly. As it turns out, the version of the open source flash firmware (OpenNFM) that I was using, while already able to easily send data across a serial cable to the controlling computer, did not have any mechanism of reliably triggering an event on the board. One option, suggested by another student, was to use a counter triggered by reads/writes performed and then use a threshold in order to trigger an action. While this suggestion would work, it wasn't practical for my needs, since I would need to take many snapshots, and would need to be able to trigger them only after I had performed actions as needed. I therefore was forced to dig deep into OpenNFM and the underlying serial communication code, to identify a method of facilitating two way communication. What I discovered was that the serial communication code for OpenNFM did contain code for reading data on the board, but the existing code was broken. I found out this code was based on a sample project for the LPC H3131 for creating a Virtual COM port (I've included this code in my submission for reference). By comparing the code, I found that the primary problems with the OpenNFM code were: first that it was not registering the proper interrupt handlers, and second that its usage of dynamic memory allocations used for its transmit/receive queues was resulting in null pointers. Registering these interrupt handlers and switching to static memory allocation lead to me a solution to this problem, which allowed to me to successfully send text data to the device. From here, I could check on the device for data sent from a computer, and then trigger an action based on it.

The next difficulty I ran into was that, while I could now successfully trigger snapshots, when I attempted to send the snapshot data to my computer, my development board was eventually unexpectedly freezing up on me, making it very difficult to take snapshots in the manner I desired. I eventually discovered that this problem was due to starvation of the read/write handler on the development board. I placed my snapshot

code in the main user loop on the device, which also handles the reads/writes, in order to ensure it would always be able to be run on demand. What I did not anticipate was that taking these snapshots would starve out the reads/writes, and that if the device is unable to handle these reads/writes for too long, the board will freeze up and prevent it from being able to act as a Mass Storage Device. My solution to this problem was to buffer out the snapshots while they print, and allow any read/write operations to occur in between, to ensure they are not starved, while still printing my snapshot effectively.

A final roadblock that presented itself was the speed of taking snapshots. I needed to take somewhere from 5 to 10 snapshots per experiment, which would mean several hundred snapshots for the data I wanted to collect. With the default settings, the device has a lot of storage space (most of which isn't necessary for my experiments), and limited capabilities in printing data to the computer, such that it could potentially take multiple days for a single snapshot to ensure that all data on the device was reliably recorded on the computer. The first optimization I made was significantly reducing the size of the flash device programmatically, by changing the number of blocks per plane from the initial value of 4096 to 128, which brought snapshot times down to closer to 20-30 minutes. I ran into additional problems with sending the raw data to my serial port monitor, because the monitor would respond to non printable characters in unexpected ways, so I switched to sending hex dumps instead. What I quickly realized was that although printing the raw flash itself was useful for verifying correctness of my transmission code, it was terrible for quickly collecting data, because I couldn't escape that I had to send all of that data across a very slow channel. A very significant optimization that struck me was the realization that I didn't need all the data, all I needed was to know whether the data was *changing*. Since the block device layer is writing at the sector level, all I needed to know was whether sectors were changing at all from snapshot to snapshot. So, instead of sending the raw data, I added the ability to send an MD5 hashed version of the data instead. I retained the functionality of sending the raw data for testing purposes, but the hash optimization had reduced snapshot times to around a minute and a half. This is because the MD5 hash (in hex) is guaranteed to be 32 bytes long, whereas the full pages I was sending before were 4096 bytes each. The probability of two arbitrary MD5 hashes colliding is incredibly low, so the concern that a sector would change but the hash would stay the same as the previous hash is very unlikely, meaning the hashes provide an equivalent determination of if a change occurred. I chose MD5 hashes (as opposed to SHA for example) because they are easy to perform, and are lightweight enough that they wouldn't come at a significant computational cost. I now was able to reliably, quickly, and accurately receive snapshots from the development board.

From here, the goal was to create a set of experiments and do analysis on them in an effective and repeatable way. Initially I was performing experiments manually, to verify that the firmware was working as intended, but I realized that it would be too much work to repeat this for each FDE scheme, and would be prone to mistakes and errors. I thus chose to write some scripts in Python, because the language happens to have some good support for serial communication, which made it very easy to create scripts to perform my experiments for me. I was also initially doing my analysis using visual diff tools such as Meld to quickly analyze what was changing. I found that tools of this sort perform poorly for my use case though, because they do not do their comparisons completely line by line. Since each of my snapshots should be the same size, and the lines should correspond to the same pages, I instead wrote an additional python script to do line by line comparisons, and then setup my experiments script so that not only does it perform the experiments and take snapshots for me automatically, but it also generates a diff between every pair of snapshots to significantly simplify analysis on the experiments.

With all of that code in place, all that remained was choosing a set of FDE schemes to use in my experiments. I chose two forks off of the TrueCrypt project, since Truecrypt is no longer considered secure, and they are open source projects. I also used a trial version of Symantec's FDE product, to include a consumer grade solution as well. I intended to also include Bitlocker in the mix, but had difficulty getting it to work with the version of Windows 7 I have, and could not get my dev board to work with Windows 10 where I had a working version of Bitlocker. I then performed 5 experiments intended to target different aspects of the problem, to see whether I could answer the question of whether FDE schemes leave structural artifacts.

Experimental Evaluation

My solution consisted of performing 5 separate experiments on 4 differently configured Mass Storage Devices. The four configurations of the mass storage device were: a plaintext configuration in which raw plaintext data could be observed on the device, configurations setup with Veracrypt and CipherShed FDE in which ciphertext data could be observed, and a final configuration using Symantec FDE which is a closed source consumer grade FDE option. The flash firmware I based my code on is the open source project OpenNFM. The firmware was ported to an NXP LPC H3131 board, so that was the development board I performed the configuration and experiments on. Each of the files used for copying and deleting in my experiments contained slightly different test text to help differentiate them, and they were all 21 KB in size (an arbitrary length). See the user/code guide for additional details on the setup and code used for the experiment.

Experiments

For each experiment, the python script I created performs a particular action (E.g. copy a file to the device) and then requests a snapshot of the state of the device. Each experiment consists of a different sequence of actions, and the results were captured and recorded for analysis. Figure 1 shows what a typical running of the experiment script looked like.

Experiment 1

The purpose of my first experiment was simply to observe what happens when a normal deletion occurs within an FDE scheme. The experiment consisted of first configuring the device, then copying three files to the device and then perform a secure deletion on the second file using window's sdelete utility. The intent of this experiment was to illustrate the effects of secure deletion within these encryption schemes.

Experiment 2

The purpose of the second experiment was to observe what happens on the device when the second file was neither copied or secure deleted. This was to establish that, counter to the Truly Secure Deletion requirements, the device state when an insert and delete sequence occurs is different from when no such insertion and deletion occurs, which establishes the existence of *some* observable structural artifact.

Experiment 3

The purpose of the third experiment was more or less a control, to illustrate the difference between a secure and non secure deletion. In this experiment three files were copied over to the device like in Experiment 1, but instead of performing a secure deletion on the second file, a normal deletion was performed.

Experiment 4

The purpose of the fourth experiment was to observe what file edits look like inside these FDE schemes. The thought was that even if experiment 1 and 2 were to establish that artifacts exists, if file edits cannot be differentiated from the deletions then the usefulness of knowing that structural artifacts exist for deletions is diminished, because if they look identical to if an edit occurred, the best you can ever say in some cases is that an edit **or** deletion occurred.

Experiment 5

The purpose of the fifth experiment was to observe what potential impact different file systems would have on the observation in Experiment 1. All four other experiments were performed by formatting the drive using a FAT file system. In this experiment, we perform Experiment 1 exactly the same, but instead we format with an NTFS file system.

Figure 1: Partial output from running the experiment python script for Experiment 1

Results and Analysis

In performing my experiments, I collected numerous snapshots and performed some processing on them to make them more useful. All snapshots collected are included with my project submission in the snapshots folder. The snapshots of most note and interest are the hash snapshots. The full snapshots are incomplete, since they take a very long time to run, they also have some irregularities in them that I haven't been able to fully explain, but I suspect are partially due to how long they take to collect. The following analysis and results are based on the hashed snapshots collected, which I believe to be the highest quality snapshots I collected. The diffs for each experiment are the most illustrative way to view the observations, but even then the amount of data in them makes it difficult to include directly in this report, so I will be referencing these diffs directly throughout my analysis. My analysis will be specifically comparing the plaintext data and CipherShed data, but very similar results are apparent in all 3 FDE schemes. From my experiments, I found that none of the FDE schemes resulted in significantly different or more difficult drive states to analyze. Refer to the User Guide for additional information on how to read and understand the snapshots and diffs.

Experiment 1 (snapshots\hash\ciphershed\experiment1\diffs)

In experiment 1 the results for the plaintext and FDE schemes were all very similar. The diffs help to illustrate the way data changed, and show that although with FDE the data is encrypted instead, with multiple snapshots we can still easily observe that a secure deletion had occurred. In particular, from the diffs s2tos3, s3tos4, and s4tos5 we can observe that file 1, file 2, and file 3 are added to the flash device in pages 80 to 90, 92 to 102, and 104 to 114 respectively. In diff s5tos6, we can see that after deleting file 2 there was a change observed in page 92 to 102. We already noted insertions occurred in pages 80 to 90 and 104 to 114, meaning we can infer that either this entire file was securely deleted or overwritten due to flash memories log structured writing. This result is the same as the case of plaintext data, the only difference is the encryption, which means we have to identify files indirectly rather than directly.

Experiment 2 (snapshots\hash\ciphershed\experiment2\diffs)

In this experiment, the key observation is that when we insert file 1 (s2tos3) and then insert file 3 (s3tos4) without doing an insertion or deletion of file 2, then file 3 immediately follows after file 1. This indicates that the drive state in the previous experiment was affected by whether file 2 existed on the drive at some point in the past. This tells us that there are structural artifacts left over by the creation and deletion of file 2, but they are difficult to detect unless you have multiple snapshots like this that allow you to watch how the data is changing.

Experiment 3 (snapshots\hash\ciphershed\experiment3\diffs)

The third experiment was meant to illustrate what the differences between a secure deletion and regular deletion are. Files 1, 2, and 3 are added from s2tos3, s3tos4, and s4tos5 respectively. A normal deletion is performed from s5tos6, and it can be observed that although the metadata pages are changed, the underlying data remains untouched. This is in contrast to Experiment 1, where the deletion affected the data itself. This is a good thing in that the data itself doesn't change meaning the deletion avoids being easily detected, but if the encryption keys are obtained or the AES encryption can be broken then the raw data is still available to an attacker. This is sort of an interesting tradeoff, because you gain security in the underlying raw flash by electing not to secure delete, but you lose security in the application level because the data isn't actually being deleted.

Experiment 4 (snapshots\hash\ciphershed\experiment4\diffs)

Experiments 1-3 are useful, because they are illustrative of the potential for leakage and structural artifacts when using Full Disk Encryption in conjunction with a conventional secure deletion solution. In Experiments 4 and 5, the goal was more to illustrate that while there is definitely potential for leakage, the significance of the leakage and its usefulness are dependent on numerous factors. In Experiment 4, the question I wanted to answer is: how different does a secure deletion look from a simple file edit? Up to snapshot 5, the 3 files are simply being added to the device. For s5tos6, s6tos7, and s7tos8, edits are made to file 2, file 1, and file 3 respectively. The edits to these files are located in different places in the file, for file 2 and file 3 the edit is made towards the end of the file, for file 1 the edit is made towards the beginning. s5tos6 shows us that edits at the end of the file only change data at the end of the file, and so as long as the file spans more than one page, edits can be discerned from deletions fairly easily. s6tos7 shows that when the edits occur towards the beginning of a file, it becomes very difficult to discern the differences between whether a file was deleted or simply edited in an insignificant way. s7to8 is interesting, because although the edit for the file is in the same location as it was in s5tos6, the diff looks very similar to s6tos7. If you look at the plaintext full snapshots of this experiment, what is actually happening is that parts of file 3 were consolidated into the block that contained file 1, causing additional pages to change. This further illustrates how many factors potentially impact what is changing between snapshots, in this case, even though we repeated a similar change, either the filesystem or flash firmware decided to store things differently making it more difficult to discern what had happened.

Experiment 5 (snapshots\hash\ciphershed\experiment5\diffs)

The previous experiments were all performed on a FAT filesystem. The advantage of the FAT file system is it is pretty predictable, so if there is leakage, it should be quite easy to observe it and learn how it behaves and how to identify it. This final experiment was to illustrate that in a more practical scenario, with the more complicated NTFS file system, it is very difficult to discern what exactly is happening. This experiment is identical to Experiment 1, but performed on NTFS instead. Starting with s2tos3, it becomes clear that even though we are only adding a single file that is 11 pages long, metadata changes results in 50 pages changing on the device. If we start with the assumption that the data is stored contiguously (this can be confirmed by viewing the plaintext full snapshot, but is not necessarily the case with NTFS), then we see there is only one block of 11 pages that changed together, so our file add occurred in pages 1872 to 1882. Similarly, in s3to4 the file add occurred in pages 1884 to 1894, and s4tos5 the add is 1896 to 1905. We can also observe the deletion in pages 1884 to 1894 in s5tos6. So, in experiment 5, we find that we **can** still discern deletions, but they are are starting to get lost in the changes to metadata, which was clearly not the case with the FAT file system. When you start thinking about an actively used file system, where changes are separated by unpredictable amounts of time and are of highly variable sizes, it becomes obvious that some additional work and analysis would be necessary to keep track of what exactly is going on.

Conclusion

In this Project, I used an open source flash memory firmware to observe what Full Disk Encryption schemes were doing below the Block Device layer, which isn't as easily done on conventional hard disk drives. I wanted to assess the feasibility of Full Disk Encryption in conjunction with conventional secure deletion as a way of removing structural artifacts, that is: do full disk encryption implementations at the application layer provide similar guarantees of data erasure that secure deletion solutions implemented at the FTL layer offer (i.e. erasure of structural artifacts). My conclusion is that Full Disk Encryption **does not** inherently provide these guarantees. That said, depending on the FDE that is used, and the file system that is used, additional confusion in the underlying ciphertext can happen between snapshots, which makes it increasingly difficult to get a good indirect picture of what is happening on the file system. Although there is some type of leakage occurring, it is very difficult to say what actual damage can be done with this leakage. In the case of Chen et al. [3], the structural artifacts in question were plaintext data, and they illustrated a scenario where contextual data could help make inferences about the data that was missing. In this case, there is very little contextual data to work with because all of the context is encrypted, so you would have to get contextual data from some type of external source. It is unclear whether an actual attack is feasible when using FDE as a solution for secure deletion, but this project hopefully helps to illustrate that such an attack is not impossible.

Some final concluding remarks are that I feel like there is a lot of potential future work branching from this project, this is in a lot of ways just the tip of the iceberg, and I wish I had been able to perform more experiments. That being said, I feel I carried out the initial goals of my project satisfactorily. I started this project without any experience in embedded programming, with limited knowledge of secure deletion and Full Disk Encryption, and limited knowledge of secure deletion. At the end of this project, I've learned considerably about using the LPC-H3131 development board, and about embedded programming in general. I've also learned a considerable amount about secure deletion and full disk encryption, and how they interact. I feel like the work performed in this project, while not necessarily earth shattering, was very valuable for my own learning, and potentially elucidates some ideas and thoughts on how FDE performs as a secure deletion solution.

References

[1] Dan Boneh and Richard J. Lipton. "A Revocable Backup System". In: *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6*. SSYM'96. San Jose, California: USENIX Association, 1996, pp. 9–9. URL: `http://dl.acm.org/citation.cfm?id=1267569.1267578`.

[2] Bo Chen and Radu Sion. "HiFlash: A History Independent Flash Device". In: (Nov. 2015).

[3] Bo Chen et al. "Sanitizing Data is Not Enough!: Towards Sanitizing Structural Artifacts in Flash Media". In: *Proceedings of the 32Nd Annual Conference on Computer Security Applications*. ACSAC '16. Los Angeles, California, USA: ACM, 2016, pp. 496–507.

[4] Niels Ferguson. *AES-CBC + Elephant diffuser: A disk encryption algorithm for Windows Vista*. 2006.

[5] *How Whole Disk Encryption Works*. Symantec Corporation, Nov. 2010. URL: https://www.symantec.com/content/en/us/enterprise/white_papers/b-pgp_how_wholedisk_encryption_works_WP_21158817.en-us.pdf.

[6] Shijie Jia et al. "NFPS: Adding Undetectable Secure Deletion to Flash Translation Layer". In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '16. Xi'an, China: ACM, 2016, pp. 305–315. ISBN: 978-1-4503-4233-9. DOI: 10.1145/2897845.2897882. URL: http://doi.acm.org/10.1145/2897845.2897882.

[7] Jaeheung Lee et al. "An Efficient Secure Deletion Scheme for Flash File Systems". In: *J. Inf. Sci. Eng.* 26 (2010), pp. 27–38.

[8] Joel Reardon, Srdjan Capkun, and David A. Basin. "Data Node Encrypted File System: Efficient Secure Deletion for Flash Memory". In: *USENIX Security Symposium*. 2012.

[9] Richard Stiennon. *Everything You Need to Know About DoD 5220.22-M Wiping Standard*. Mar. 2019. URL: https://www.blancco.com/blog-dod-5220-22-m-wiping-standard-method/.

[10] *Veracrypt Technical Details*. Veracrypt. URL: https://www.veracrypt.fr/en/Technical%20Details.html.