Full disk encryption on unmanaged flash devices

Matthias Fabian Petschick
(matthias@net.t-labs.tu-berlin.de)

September 27, 2011

Technische Universität Berlin
Fakultät IV
Institut für Softwaretechnik und Theoretische Informatik
Professur Security in Telecommunications

Betreuender Hochschullehrer:	Prof. Dr. Jean-Pierre Seifert
Zweitkorrektor:	Prof. Dr. Hans-Ulrich Heiß
Betreuender Mitarbeiter:	M.Sc. Collin Mulliner

Erklärung

Die selbständige und eigenhändige Anfertigung versichert an Eides statt

Berlin, den 27. September 2011

Matthias Fabian Petschick

Abstract

This thesis deals with the encryption of unmanaged flash devices, typically found in mobile devices like smartphones nowadays. While there is a multitude of software available that deals with this task in the context of managed flash and other block devices, the lack of existing solutions for this type of hardware makes it hard to ensure confidentiality of the data stored on it. With the growing market share of smartphones, the urgency to find an appropriate method to make up for this deficiency is rising.
In this thesis, we discuss our approach at tackling this problem. We describe the design and implementation of a cryptographic layer that integrates into Unsorted Block Images (UBI), an existing management system for Memory Technology Devices (MTDs) present in the Linux kernel. We outline obstacles and explain our design decisions, based on which we a develop a prototype implementation. This implementation is tested on the Nokia N900 smartphone, for which we provide an extensive evaluation of the test results. The main focus of attention lies on mobility-related aspects such as power consumption and durability as well as file system performance and system load. Based on this, we review our initial assumptions and analyze how well they are met. Furthermore, we discuss problems and questions remaining open as well as possible solutions for them. Finally, we draw comparisons to related projects, provide an outlook on future work and present the conclusions we draw from our work.

Kurzbeschreibung

Diese Diplomarbeit beschäftigt sich mit der Verschlüsselung von Unmanaged Flash Devices, die heutzutage typischerweise in Smartphones zu finden sind. Während eine große Menge an Software verfügbar ist, die sich mit der Verschlüsselung von Managed Flash und Block Devices beschäftigt, existiert ein Mangel an Konzepten für Unmanaged Flash, was es erschwert, die Geheimhaltung der darauf gespeicherten Daten zu gewährleisten. Mit dem wachsenden Smartphonemarkt wächst auch die Dringlichkeit, eine passende Lösung für dieses Manko zu finden.
In dieser Diplomarbeit diskutieren wir unseren Ansatz, mit dem wir uns mit dieser Problematik auseinandersetzen. Wir beschreiben das Design und die Implementation eines kryptographischen Layers, welcher in Unsorted Block Images (UBI), ein bestehendes Managementsystem des Linuxkernels für Memory Technology Devices (MTDs), integriert wird. Wir legen die damit verbundenen Schwierigkeiten dar und erklären unsere Designentscheidungen und Vermutungen, welche wir als Basis für die Entwicklung eines Prototyps verwenden. Dieser wird auf dem Nokia N900 Smartphone getestet und die dabei entstehenden Ergebnisse werden evaluiert. Das Hauptaugenmerk liegt dabei auf Mobilitätsaspekten, wie Stromverbrauch und Lebensdauer, sowie auf der resultierenden Systemauslastung und der Geschwindigkeit von Dateisystemzugriffen. Darauf basierend betrachten wir, inwieweit unsere Ausgangsvermutungen zutreffend sind. Des weiteren diskutieren wir noch offene Fragen und Probleme, sowie mögliche Lösungsansätze. Abschließend ziehen wir den Vergleich zu verwandten Projekten, geben einen Ausblick auf zukünftige Arbeit und präsentieren die Schlüsse, die wir aus unserer Arbeit ziehen.

Contents

1. Introduction

With today's increased presence of smartphones both in the business and private sector, the development of better methods to ensure the confidentiality of the data stored on mobile devices has become more and more urgent. While there are many products on the market that each focus on encryption of specific areas on these devices, there is a distinct lack of solutions that deal with unmanaged flash. This type of memory is frequently used in all kinds of embedded devices, ranging from home routers to smartphones. Since it is not always clear to the user how applications and operating system (OS) store their data, there is a risk of confidential data ending up on unencrypted flash memory. In this thesis, we take a look at different approaches to prevent this and analyze why encryption on unmanaged flash devices is not as widespread as it is on managed ones. In the following, we present the design, implementation and evaluation of a solution aimed at eliminating this deficit. Since security rarely comes without a cost, it is important to know the price in advance. It is one of our goals to give the reader an idea of what to expect by pointing out where the pitfalls really are.

To emphasize the real-world applicability, our prototype was developed for and tested on the Nokia N900, a smartphone known for its open platform called Maemo which is based on the Debian Linux distribution. This thesis discusses all necessary steps for a full encryption of all data stored in non-volatile memory on the N900. Both advantages and disadvantages of our approach are examined and weighed against each other, backed by measurements performed using well known tools and benchmarks as well as self-written ones. This includes filesystem and power consumption benchmarks as well as experiment setups investigating system load and flash endurance. An overview over related work and an outlook on future work is given. Last but not least, the conclusions that we draw from our work are presented.

1.1. Contributions

The main contributions of this thesis are

- **A working encryption system for unmanaged flash devices**
  Unmanaged flash devices are often found on mobile devices such as smartphones. One key feature of this system is that it is not limited to a specific filesystem, unlike other existing solutions.
- **An analysis of our system under various aspects and a prototype implementation**
  Because resources on the target platforms are limited, we review our system under this aspect. We look at various advantages and disadvantages of our approach and use a prototype implementation to confirm our assumptions.

- **A performance evaluation of the prototype**
  We provide detailed measurement results and evaluations in order to assess the viability of our solution. To put our results into perspective, we also draw comparisons to related approaches and point out the major differences.

1.2. Thanks

At this point we would like to thank Nokia for providing a free N900 to support the development for this thesis. We also received competent support from Nokia technicians regarding hardware specifications, drivers and most importantly a boot loader to enable the crypto co-processor, some of which were not freely available at the time of writing.

2. Background, motivation and goals

In the following, we take a closer look at the problems this thesis tries to solve and the motivation behind it. Naturally, this requires a bit of background information for better understanding, which is provided as well. Based on this, we can outline the actual goals in order to determine later in how far they were achieved.

2.1. Background

As this thesis deals mainly with smartphones and cryptography, the next sections explain how these two relate and why this is important to us. We also provide some background information about tools used for measurement and evaluation of our work.

2.1.1. Smartphone ubiquity

Despite the fact that feature phones still dominate the market, smartphone sales are rising and have increased by 50.5% in the second quarter of 2010, compared to the same period one year earlier [1]. Since then, a new competing smartphone OS has joined the market and can be expected to further push the scale towards smartphones in regard to market share. A survey done by RingCentral, a provider for cloud computing based phone systems, suggests a growing importance of smartphones in the small to medium-sized business sector in the United States [2]. It is not unlikely that many of these devices contain work-related confidential information, for example in the form of emails, notes and schedules. Likewise, phones used privately may contain data that the user would prefer to be kept secret. Due to their size and typical use, smartphones are often carried around on person and thus face the risk of being lost or stolen. Apart from the possibility of these mishaps causing large phone bills, there is the non-negligible danger of confidential data being revealed. In some cases, this may lead to bigger headaches than a few long distance phone calls could cause. Subscriber identity module (SIM) cards can be blocked by calling the provider but access to the device memory normally does not depend on network connectivity or knowledge of the personal identification number (PIN). It is therefore vital to find means for ensuring the protection of all private information stored on smartphones. There are multiple attack vectors, such as data being stolen by malicious software or physical access to the device in an unlocked state and covering them all would be well beyond the scope of this thesis. Instead, we focus on preventing access to the device memory after it has been turned off. A typical approach is to employ strong cryptographic ciphers to encrypt everything stored on it.

Cryptography and smartphones

Cryptography is no new concept in the field of mobile communication. All in all, a lot of effort is done to secure the communication between phone and network provider. Since this thesis focuses on the data stored on phones, this raises the question of what is done to protect user data on the phone itself. This is not a task for the provider but for the company developing the phone's OS or third party developers. Newer versions of both iOS (previously known as iPhone OS) and BlackBerry OS support encryption of data stored locally [3], [4]. For Android, a third party application supporting root filesystem encryption by WhisperSystems [5] became available during the course of our work on this thesis.

However, some of these solutions are not without limitations [6]. Others are threatened to be intentionally weakened due to political pressure [7], [8]. Due to the nature of proprietary solutions, the user is required to put a certain level of trust into the respective company. Moreover, many smartphones and smartphone OSes rely on unmanaged flash, also called raw flash.

Unmanaged flash memory

Unmanaged flash memory differs from managed flash memory which is usually found in consumer flash devices like MultiMediaCard (MMC), Solid-state drives (SSDs), Universal Serial Bus (USB) flash drives, etc. While the latter comes with an integrated controller that takes care of wear leveling, garbage collection and other tasks typically related to flash memory, the former requires these tasks to be handled by the OS. Additionally, raw flash devices provide low level information such as ECC (Error-correcting code) and allow control over the location where data is stored on the chip. Managed devices hide the details of where and how data is stored from the OS and instead provide a simple block device that can be used just like a hard disk drive. The advantage of unmanaged flash lies in its flexibility and lower price. Giving more control to the OS also allows the use of filesystems which are optimized for use on flash memory. These filesystems can leverage the advantages flash has over hard disk drives, resulting in improved performance and durability. The latter refers to the limited number of program-erase (P/E) cycles which all types of flash memory inherently possess. Each of these cycles decreases the remaining lifetime of the cells in a physical erase block (PEB). Samsung for example guarantees 100,000 P/E cycles without device failure caused by an increase of bad blocks for its OneNAND chips [9]. While a managed flash controller can optimize all accesses to the memory, it is neither aware of the specifics of the filesystem being used nor is the filesystem itself aware of the memory properties. In contrast, optimized filesystems such as Yet Another Flash filesystem (YAFFS) and UBIFS (Unsorted Block Images File System) are designed to take advantage of the properties of unmanaged flash. There is however a general disadvantage that all types of flash memory exhibit when it comes to encrypting them. The following section explains this in more detail.

2.1.2. Cryptography and flash memory

As mentioned before, flash memory has a few special properties, some of which are disadvantageous when it comes to applying typical full disk encryption (FDE) methods. Before this is described in more detail, it is important to be aware of how data is actually written to and read from flash memory. On the lowest level, information is stored in cells, which are made up of transistors. For *Not And* (NAND) chips, the smallest unit of access to these cells available to the OS is a page[1]. Page sizes differ among various flash chips and typically range between 512 and 4096 bytes, or multiple cells. *Not Or* (NOR) chips allow finer grained access[2] but both types of flash memory have one property in common: all bits are initially one and after being set to zero cannot be reverted back to one, unless a larger area or block they belong to is erased, which changes all bits in this block to one. This block, also called PEB, consists of multiple pages and is comparatively large: 64 to 256 KB on NOR memory and 16 to 512 KB on NAND memory. Since the erasure of such a large block is a costly operation performance- and durability-wise, flash filesystems such as UBIFS usually perform it in the background and simply move the modified block content to an empty block during an update operation. With this knowledge, it becomes very easy to tell empty and used blocks apart: the former ones contain solely 0xFF bytes[3]. From a security point of view, it is desirable to reveal as few information about the protected data as possible and the ability to differentiate between empty and full space can leak considerable amounts of information [11]. A typical approach for full disk encryption to prevent this is to fill up all empty space with pseudo-random data. Doing so on a flash memory chip results in a very undesirable situation: since there are no empty blocks left, it becomes necessary to erase a full block for even the smallest update and wait for this operation to complete. Consequentially, both performance and hardware life-time are expected to be negatively affected.

2.1.3. Hard disk encryption background

Hard disk encryption has been around since more than a decade and many different methods and tools have been developed over the years. Some very prominent software implementations are PGPDisk, Truecrypt and dm-crypt. Hard disks are usually block oriented devices and thus block ciphers such as Advanced Encryption Standard (AES) are the most commonly employed, typically in combination with a corresponding cipher mode of operation, for example Cipher-Block Chaining (CBC). Block ciphers on their own provide a mapping between the plain text and cipher text of a block of fixed size n, often 64 bytes, dependant on an encryption key k. While it is possible to split a longer message into n-sized blocks and apply the encryption function E_k to each of them, this method, also called Electronic Codebook (ECB), is commonly not used as it does not hide data patterns [12]. A more commonly used method of operation is the previously mentioned CBC. This algorithm uses two adjacent blocks which are combined by XORing the cipher text of the previous block with the plain text of the current block and subsequently encrypted. Obviously, since there is no previous block for the first block, this method requires an initialization. This is achieved by supplying an initialization vector which is used in place of the non-existing block previous to the first block. In the context of block devices, which are generally organized in sectors,

[1]Single Level Cell (SLC) NAND also supports a limited number partial programmings [10], dividing a page into even smaller chunks.

[2]NOR chips provide random read and write access at a minimum Input/Output (I/O) unit of 1 byte

[3]This disregards block management information which is usually stored at the beginning of each block by the filesystem. Since it makes up only an extremely small part of the whole block and due to its well-known location, it does not matter here.

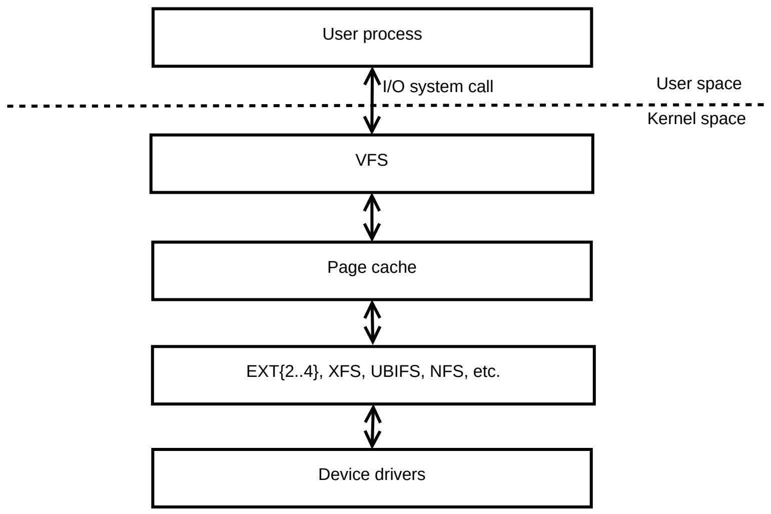


Figure 2.1.: VFS and I/O caching

the sector number can be used as initialization vector (IV). This results in a unique IV for each sector, however it makes the IV predictable and thereby vulnerable to so-called watermarking attacks. This type of attack allows an adversary to detect and prove the existence of specifically marked files on the encrypted medium. Clemens Fruhwirth developed an IV scheme called Encrypted Salt-Sector Initialization Vector (ESSIV) aimed at preventing this type of attack by making the IV unguessable [13]. To generate an IV, this algorithm encrypts the sector number with a salt that is generated from the hashed encryption key. There are however further weaknesses to be found in CBC-ESSIV, such as content leaks, malleability and modification detection [14]. These attacks require that the attacker has either access to an older version of the encrypted data or that he has access to the medium while it is in use. There are other modes of encryption which do not suffer from these vulnerabilities, such as Lislov, Rivest and Wagner (LRW), a tweakable block cipher. LRW provides better security, however this comes at a performance cost. Nevertheless it was considered in a draft for the IEEE-P1619 standard for storing encrypted data, at least until some additional weaknesses were discovered [15]. It was superseded by Xor-Encrypt-Xor (XEX)-based Tweaked CodeBook (XTS) in the final version of IEEE-1619 [16].

2.1.4. Caching

While it is not the only important aspect, it is obvious that measuring filesystem performance plays an important role in our evaluation. As a result, we have to deal with caching. Let us take a short look at why this is the case. All file system operations in the Linux kernel are tied to the Virtual Filesystem Switch (VFS). The VFS is basically a software layer which handles all filesystem related system calls and provides a common interface to several kinds of filesystems [17]. As seen in Figure 2.1, all data passing between VFS and the actual device typically goes through the so-called page cache[4]. The page cache is generally used to speed up read and write accesses by mapping page-sized file chunks into memory. If, on the one

[4]There is of course an exception to this rule called direct I/O, which allows the page cache to be bypassed by applications that prefer to implement their own caching algorithms.

hand, a process wants to read data which is already present in the cache, the kernel can return it without waiting for the much slower device the related file is stored on, for example a network device. For write access on the other hand, data is updated in the page cache, from where it is saved to the underlying medium at a later point, which is also referred to as deferred writing. The process does not need to wait for this comparatively slow operation to complete and the write system call returns the number of bytes written as soon as the kernel has finished copying the data from user memory to the page cache. Of course, data stored in main memory can become subject to data loss, for example during a power loss, therefore the modified pages are written to disk in regular intervals or if the cache runs out of free pages. Optionally, a process can request a file to be opened for synchronous write access, which causes write calls to wait until the data has been transferred to disk. Additionally, the flushing of all cache entries related to a file handle to disk can be explicitly requested.
The VFS also uses two separate caches for i-nodes and directory entries to speed up directory and file lookups, called the inode and dentry cache[5]. Why is all of this important to us? The answer is quite simple. One obvious method of measuring the performance of our implementation is running filesystem benchmarks on top of it. In order to be able to correctly parameterize our benchmarks in regards to the effects of filesystem caching and to ease the interpretation of our results, it is helpful to be aware of how caching works.

2.1.5. Measurement tools

We already mentioned the importance of filesystem benchmarks in Section 2.1.4 and thus it should come without surprise that two of our main benchmarking tools are indeed filesystem benchmarks. We also want to know where bottlenecks lie for potential future improvements. For this task we need a so-called profiler, a software that monitors, amongst other things, the resource usage by the OS, itself or other processes, as well as the time spent in function calls and the frequency of these calls.

- **iozone**
  Iozone is a filesystem benchmark that can be tweaked in many ways to simulate various use-cases and application behaviors, for example it can be tuned to re-enact the typical file access patterns of a database application. It reports detailed information and provides several options to control benchmarking parameters. Iozone is available for various platforms, although we mainly care for the Advanced RISC Machine (ARM) platform.
- **bonnie++**
  Bonnie++ is yet another filesystem benchmark. It is a reimplementation of the bonnie hard drive benchmark by by Tim Bray, with new features added such as support for storage sizes beyond 2 GB and testing operations involving thousands of files in a directory [19]. It is less versatile and provides fewer details in its output than iozone. We use it for comparative measurements to confirm trends observed with iozone and to generally have a ″second opinion″ available.
- **oprofile**
  Oprofile is a profiler that is capable of profiling any code running on a Linux system,

[5]Refer to [17] for more details

both in user and kernel mode. Some of its features are, according to the authors [20], unobtrusiveness, low overhead and call-graph support. In addition to the ARM platform support, these features make it our tool of choice for analyzing the performance of our prototype in detail. Oprofile relies on hardware performance counters[6] and consists of a daemon and a loadable kernel module that comes with all Linux kernels since 2.6. After the daemon is started, it records its measured data in a temporary filesystem (tmpfs) directory by default. This data can later be used to plot call-graphs of the functions called during the profiling period.

- **HAL**
  The N900 battery includes a pin which can be used to determine the installed battery's charge level. Access is provided by a kernel driver and a Hardware Abstraction Layer (HAL) plugin. We can use the hal-device tool to query HAL about current battery charge levels, which we can use to measure power consumption on the N900.

2.2. Problems and motivation

In the previous section, a number of problems in relation to confidentiality of data stored on smart phones were mentioned. Some of them lie out of scope, therefore it is important to highlight those that are addressed in this thesis.

- **Designing a system that protects data on unmanaged flash memory**
  While managed flash is comparatively easy to encrypt using widely available software for block devices like dm-crypt, unmanaged flash is often unprotected. A system suitable for its encryption must be designed, preferably one that integrates seamlessly into an existing device, e.g. a smartphone.
- **Ensuring compatibility with existing file systems**
  Existing flash-aware filesystems should be able to run on encrypted flash or be portable without complex modifications. While the device-mapper[7] in the Linux kernel allows completely transparent access to encrypted block devices, the situation is more intricate with flash-aware filesystems. It has to be determined in how far the assumptions made by them regarding flash devices are compatible with encrypted devices.
- **Implementation and evaluation**
  Proving that the design works in practice and measuring how well it performs is essential for being able to make any meaningful statements about applicability in the real world. Without actual numbers, it is hard to determine the trade-offs. To produce them, a basic implementation that allows measurements of the primary aspects of our solution is required.
- **Keeping things simple**
  While the long-term goal of this work is a versatile implementation providing maximum compatibility, the limited time frame of a diploma thesis forces us to focus on

[6]Hardware performance counters are an optional feature provided by many CPU types to keep track of various things like the absolute number of cache misses, the number of instructions issued or the number of floating point instructions executed [21].

[7]The Device-mapper is a component of the 2.6 Linux kernel that supports logical volume management [22].

the aspects required to prove or disprove our assumptions and does not allow us to flesh out all the details. Keeping things as simple as possible works towards this goal. We provide a proof of concept implementation necessary to deliver the measurement results. Open questions and problems as well as ideas for future improvements can be found in Section 7.

- **Putting it into perspective**
  Depending on the results from the previous point, both pros and cons of the presented solution must be weighed against each other. Apart from performance, other factors such as flexibility and usability play important roles as well. Their level of importance may differ depending on the priorities of the end user, therefore it is important to review them under different perspectives.

3. Design and metrics

In the following we describe our approach to the problems discussed in the previous section. We present our design options for this task and explain our choice. This includes an introduction to the development platform used to test our implementation, details about the filesystem and device layer and about our integration of cryptography between the two. We finish with a description of the problems related to this integration and a discussion of our choice of metrics.

3.1. Development platform

Our development platform of choice is the Nokia N900 smartphone. It possesses a few features that make it the perfect candidate and which can only be found in subsets elsewhere.

3.1.1. Why the N900

The N900 is a smartphone released by Nokia in November 2009 and comes with the following features relevant to us:

- Debian Linux based OS
- Texas Instruments (TI) OMAP 3430 chipset
- UBIFS as root filesystem
- MMC card slot
- Hardware keyboard
- Software Development Kit (SDK)

The OS running on the N900, namely Maemo 5, suits our purpose extremely well due to being based on the Linux kernel and its resulting openness. Standard knowledge of Linux driver development can be applied, simplifying the development process. The ability to install custom kernels and modules using the built-in package management system lowers the barrier for user adoption. Furthermore, the Linux kernel provides many facilities like the crypto application programming interface (API) and the MTD subsystem which play an important role in our design. While the N900 is not the only smartphone on the market that uses a Linux based OS, its root filesystem is one of the distinguishing features and a very important factor in our decision. In particular, the concept of separating filesystem and volume management tasks into two separate layers, UBIFS and UBI, helps us with our goal of having a cryptographic layer that is as independent from the filesystem as possible. Last but not least, the OMAP 3430 chipset used by the N900 comes with a cryptographic coprocessor which can be used for accelerating and offloading the cryptographic operations.

3.1.2. Hardware specifications

The N900 is based on the TI OMAP 3430 chipset which integrates the ARM Cortex-A8 microprocessor, running at 600 MHz. This CPU is commonly found in smartphone-type devices [23] at the time of writing. The OMAP 3430 includes a cryptographic co-processor which supports 128-bit AES with the cipher modes of operation CBC and ECB. The Secure Hashing Algorithm (SHA-1) and Message-Digest Algorithm (MD5) are supported as cryptographic hashing algorithms. There is a total amount of 256 MB RAM available. Furthermore, the N900 features 256 MB of internal OneNAND[1] storage for the root filesystem and an internal 32 GB MMC card intended for user applications and data. In addition to that, a slot for micro Secure Digital High Capacity (microSDHC) cards is installed. This could easily be used for secure key storage, either in addition to a passphrase or as a replacement. The N900 also features a hardware keyboard which is much better suited for passphrase entry than a virtual touchscreen keyboard, as it allows faster typing and easier concealment of the typed characters.
Other available features typical for a smartphone are a resistive touchscreen, bluetooth and 802.11 wireless radio. Further details can be found at [24].

3.1.3. Software

As mentioned above, the N900 ships with Maemo 5 which is based on Debian Linux and was developed by Nokia for use on internet tablets and smartphones sold by the company. At the time of writing, the default kernel version in use is 2.6.28. The Linux kernel provides an interface for adding cryptographic hardware drivers called crypto API which makes it easy to toggle between a software implementation of various cryptographic algorithms and a hardware driver, if available. Using the drivers provided by Nokia for the OMAP 3430 AES and SHA implementation, we can take advantage of this feature. There is an SDK available which comes with a QEMU-based emulator, giving us the chance to test some of our code before running it on the actual device. The 256 MB unmanaged OneNAND flash chip on the N900 is by default split into six partitions.

Number	Name	Content
0	boot loader	the boot loader
1	config	configuration data in a proprietary format
2	log	reserved for kernel error logs in case of a kernel panic
3	kernel	the kernel image
4	initrd	unused and apparently reserved for an initial ram disk (initrd)
5	rootfs	root filesystem

The rootfs partition makes up the largest part of the whole device with a total of 227 MB. It is managed by UBI and contains exactly one UBI volume, which is formatted as UBIFS and contains the actual root filesystem. Aside from the NAND chip, the N900 also comes with a 32 GB built-in MMC card which contains the user home formatted as the Third Extended Filesystem (ext3), a large folder called *MyDocs* that is mounted into the user home

[1]OneNAND a flash chip produced by Samsung that, according to [9], combines the advantages of regular NAND and NOR memory: fast sequential and fast random access.

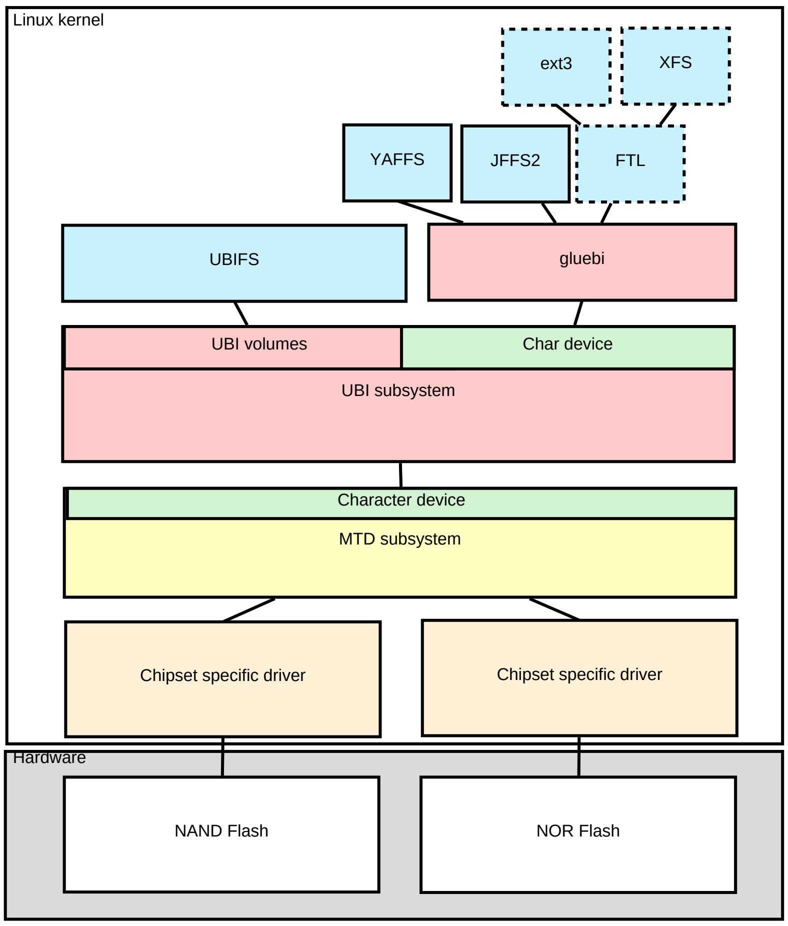


Figure 3.1.: UBI

and is formatted as File Allocation Table (FAT) and a swap partition. Most likely due to its size and widely supported filesystem, *MyDocs* is also used to share files with computers the N900 is connected to via USB. Our main focus is the partition managed by UBI because it cannot be encrypted as easily using dm-crypt as the other partitions containing traditional block-based filesystems. We ignore the configuration partition because its content is in a proprietary format unknown to us. It is impossible to encrypt the boot loader and kernel partitions, although the kernel can theoretically be stored on a removable medium for additional security. Since kernel error logs can contain confidential information, it may be of interest to encrypt the third partition as well. This requires a patch to the *mtdoops* kernel module which we do not cover in this thesis. As a simpler protection method, it is also possible to remove the *mtdoops* module from the kernel command line and thus prevent errors from being logged.

3.1.4. UBI and UBIFS

UBIFS is an open source flash filesystem developed by Nokia. It is in use on both the Nokia N800 internet tablet and the N900 smartphone. Unlike other flash filesystems, UBIFS cannot operate directly on MTD devices but instead depends on the UBI subsystem, as seen in

Figure 3.1. UBI is responsible for volume management and wear-leveling and provides a higher abstraction of MTD devices called UBI volumes. Each volume consists of a number of Logical Erase Blocks (LEBs) which UBI maps to PEBs on the flash medium. Through an extension called *gluebi* which emulates an MTD device on top of an UBI volume, it is possible to run other flash filesystems such as JFFS2 on top of UBI. Standard block based filesystems such as ext3 or XFS are not supported directly, however a flash translation layer (FTL) could be implemented as a remedy. There is no known implementation to date but the concept is described in detail on the Linux MTD discussion list [25]. Even without support for block based filesystems, UBI together with *gluebi* provides a tremendous level of flexibility and plays towards our goal of supporting existing filesystems without requiring complex modifications. By extending UBI with a cryptographic layer, no modifications at all should be necessary for file systems that can run on UBI volumes. Before explaining the layer itself, we need to take a short look at how UBI interfaces with MTD devices. In principle, UBI bears a lot of similarity with the Linux device-mapper. In particular, they both provide an abstraction of the underlying device as volumes to the upper layers. An important difference is the subsystem they interface with. Unlike the device-mapper which operates on the block layer in the kernel, UBI uses MTD as a backend, which uses character devices instead. Even though MTD is aware of and provides information about the organization into erase blocks on the underlying device, its interface provides only functions to read and write arbitrary amounts of data at specified offsets. This means that any code using this interface is free to choose whether to ignore this block based organization or not. Due to the nature of flash memory, the block sizes managed by MTD and the device-mapper can differ significantly. Historically, sectors on hard drives have a size of 512 bytes and even though newer hard drives have started shipping with 4 kilobyte sectors, the device-mapper still operates on 512 byte sectors (kernel version 2.6.38). As explained before, the smallest unit of access for writing and reading flash memory can be between 1 and 4096 bytes. The MTD interface provides this information for each of its managed devices and UBI uses this to determine the smallest I/O unit for its operation[2].

3.2. Cryptographic considerations

Even though the MTD subsystem is not part of the kernel block layer, the data on the adjacent layers is organized in blocks, albeit of a different size. This makes it possible to apply block ciphers in this context. As mentioned previously, there are various block cipher modes of operation in use for protecting data on block-oriented storage devices. Two modes commonly used are CBC in combination with ESSIV and more recently XTS. The latter is specified as the standard method for this purpose by the Institute of Electrical and Electronics Engineers (IEEE) [16]. To give the choice to the user which mode to use, it is desirable that both cipher mode and mode of operation are selectable during runtime instead of being compiled into the module. The Linux crypto API interface makes this comparatively easy to implement. However, for our prototype implementation we limit ourselves to CBC with ESSIV. The main reason for this is the available support for this mode of operation by the N900 cryptographic co-processor. This gives us the ability to directly compare a hardware accelerated solution

[2]For NAND chips that support sub page reads and writes, also called partial page programming [26], UBI uses the minimum partial programming size for its headers, which is a fraction of the page size.

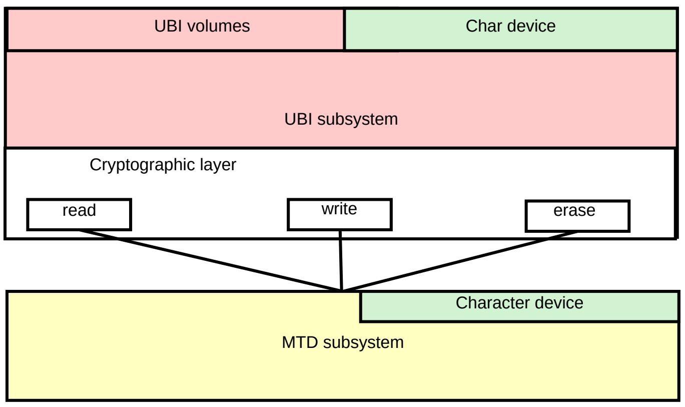


Figure 3.2.: UBI cryptographic layer

to one implemented in software. While XTS would be the preferable method in terms of security, CBC is sufficient for proving the validity of our design. One additional problem that comes with this choice is related to the recursive dependency of consecutive cipher blocks. Whenever one block changes, all subsequent blocks are changed as well. We will come back to this in Section 3.3.2.

3.3. The UBI cryptographic layer

As described above, adding cryptography to UBI gives us the flexibility we wish to achieve. The MTD subsystem provides an abstraction layer between flash memory drivers which handle the chip specifics and kernel code that wants to access this memory. It furthermore supports partitioning of the available memory and maps each partition to /dev/mtd*X* as character devices, *X* being *0* to *number of partitions*. Apart from this userland interface, a number of functions are available in kernel mode that provide I/O access, block erasure and locking, amongst others. There is a very limited number of places where UBI actually uses them, which is good for us because it reduces the amount of necessary modifications. The general idea is to place a wrapper around these function calls that encrypts or decrypts all data passing through it, respectively (also see Figure 3.2 in this regard). This ensures complete transparency for UBI regarding whether data is stored encrypted or as plain-text on the underlying device.

3.3.1. PEB numbers and IVs

Most cipher modes of operation rely on the sector number for generating IVs when used on hard disk drives. The equivalent of the sector number for UBI devices is the PEB number. In CBC mode, the ciphertext of a block relies on all previous blocks up to the IV, that means if one block is modified, all following blocks relying on the same IV must be updated. Because PEBs are a lot bigger than sectors, using the PEB number as IV would cause a large number of updates whenever a block at the start of a PEB was updated. To reduce this overhead, we split each PEB into smaller sub blocks of a size that equals the minimum I/O size of the respective UBI device for IV generation. We extend the 32 bit PEB number to 64 bit and use the most significant bits for the PEB number and the least significant bits for the respective sub block index.

3.3.2. Complications and more design decisions

As we hinted earlier, there are a few factors that complicate the design of the aforementioned cryptographic layer. UBI relies on a few properties that are given for any kind of flash, for example the existence of erase blocks and the fact that empty blocks contain solely 0xFF bytes. There are basically three possible approaches to deal with this fact

- Leave unused space untouched
- Fill it with pseudo-random data that translates as 0xFF bytes to UBI
- Implement a system of handling empty space that does not rely on 0xFF bytes

The first option leaves the risk of revealing information to an attacker. This includes but is not necessarily limited to filesystem meta information that can be deduced from the layout of the encrypted data, hints about the filesystem being used and the mere fact that the medium is encrypted. Clearly, its major advantage is the lowest amount of management and I/O overhead from leaving unused space untouched. The third option needs a bit more explanation to understand why we do not choose it. As we explained, our cryptographic layer is placed between UBI and MTD layer, while it is technically part of UBI. Keeping things as simple as possible is one of our mentioned goals, therefore it is preferable to keep the number of required modifications to UBI at a minimum. Unfortunately, many functions in UBI rely on the assumption that empty space contains 0xFF bytes, therefore it would be necessary to add an additional virtual layer for keeping track of empty space within UBI and either filter these functions or modify them appropriately. This is a lot more complicated than the second option, however it can be used to minimize the overhead from filling up empty space by intelligently managing this empty space, for example by adding buffer caching to UBI. We choose the second approach for our prototype implementation as it provides an equal level of security as the third approach, and is comparatively simple to implement. It allows us to compare security gain and overhead more directly without regarding special cases resulting from the third option. The translation mechanism of pseudo-random data is simplified for the sake of this thesis. We simply encrypt 0xFF bytes in place of empty space that, when decrypted, automatically fulfill the expectations by UBI. This approach requires the use of a mechanism to prevent watermarking attacks. A common method for this kind of problem is using an ESSIV in conjunction with a block cipher mode of operation like CBC. Alternatively, XTS mode can be used but since the N900 does not provide hardware acceleration support for XTS and due to lack of time, we do not cover this in more detail here. With the requirement of empty space being encrypted, it becomes necessary to initialize an empty MTD partition prior to using it with UBI. Initialization in this context means that every block must be encrypted and written once. Consequently, this method can also be employed to convert an existing unencrypted partition to an encrypted one. The obvious downside is, again, the additional overhead from encrypting the empty space alongside the data.

An additional disadvantage of our chosen approach is that it can be expected to wear out the flash faster than usual. Technically, there is no empty space on the flash medium and we are forced to erase a full PEB whenever we want to write to it, even if we are writing only a fraction of its actual size. Since we are using CBC, we are also forced to decrypt and re-encrypt blocks adjacent to the written data if the write request was unaligned or overlapped into the next sub block which depends on a different IV. This last problem can be alleviated by using XTS, which works without block chaining.

3.3.3. Key management

Key management is an integral part of designing a cryptographic system. It is important to ensure that the encryption key or any other secret information is not leaked to an attacker during the encryption setup phase or at a later point. If the root filesystem is to be encrypted, it is also important to make sure that a possibility for the user to enter his passphrase during the system boot exists. While some parameters can be passed though the kernel command line, it is certainly not advisable to use this for confidential information. Instead, they can be queried from the user from an init script. There are various methods commonly used, such as passphrases, fingerprint scans, keys placed on removable storage or smart cards. For the scope of this work, we limit ourselves to describing the easiest of these methods: passphrase entry by keyboard. It should be trivial to add support for keys stored on the removable MMC card that can be used with the N900.

3.3.4. Recognizing encrypted devices

When an MTD device is attached to UBI, it is necessary to distinguish between three different cases:

- Device is encrypted
- Device is unencrypted and to be mounted as-is
- Device is unencrypted and to be converted to encrypted device

Unless encrypted devices are specifically marked, it is hard to automatically distinguish between encrypted and unencrypted devices. A scheme similar to the one implemented by Linux Unified Key Setup (LUKS)[3] headers could be employed here. First, this prevents plausible deniability[4], second it is sufficient for our prototype implementation to be able to pass the information necessary to recognize encrypted volumes manually. Like the encryption parameters, this can either be done using *ubiattach* or the module parameters.

3.4. Measurement metrics

Now that we have outlined the basics of our design and its related problems, we need to find appropriate metrics to verify our assumptions. These metrics depend on a set of factors that we consider most relevant when evaluating our work.

[3]LUKS is an implementation of TKS1 [18], which specifies a cryptographic key setup scheme

[4]Plausible deniability in cryptography refers to denial of the existence of an encrypted volume and the inability to disprove this claim.

3.4.1. Metrics

There is a total of three factors that interest us. All of them play an important role in the decision of whether our solution is suitable for a particular case or not.

- **Performance**
  The task of encrypting and decrypting all data that is processed during filesystem accesses causes additional load on the CPU, unless a cryptographic co-processor is used for off-loading. Depending on the underlying hardware, this may have significant impact on the overall system performance. While filesystem encryption in general has often been analyzed in the context of traditional server and desktop systems, there are less papers on filesystem encryption on embedded systems. Multi-core processors, as they have become more prevalent in desktop and server systems lately, are well suited for handling this, however system-on-a-chip (SoC) packages for smartphones or tablet devices using multiple CPU cores like the ARM Cortex-A9 have only recently become available, like the Apple A5 in March 2011 [27]. This means that all running processes need to share this single resource, increasing the impact from filesystem encryption. It needs to be determined which percentage of overall system load is caused by this and in how far the responsiveness of the phone is affected, for example when starting applications. For this purpose, we need two metrics: CPU load and execution time. CPU load is generally measured in seconds of processing time while response time reflects the time required to execute typical tasks on a smartphone, like starting the web browser, or installing applications. Furthermore, we want to get an idea of what kind of performance improvements we can expect from using the cryptographic co-processor on the OMAP 3430. For this, we additionally measure the raw throughput of both software and hardware implementation of a specific cipher in kilobytes per second. Last but not least, we want to see how filesystem performance is affected. This can be measured in throughput in kilobytes per second and accesses per second.

- **Effects on power consumption**
  Smartphones normally run on battery which makes it important to examine power consumption as well. Additional CPU cycles or use of the cryptographic co-processor cause the battery to be drained faster, decreasing the time until the phone needs to be recharged. Consequently, the metrics we use here are maximum uptime in seconds and maximum number of jobs that can be completed on one battery charge.

- **Flash endurance**
  We have already pointed out the implications of causing additional P/E cycles. Consequently, analyzing the effect of our cryptographic layer on the endurance of the flash chip is an important aspect. This is especially important with our chosen method of handling empty space on the device. The main questions here are by how much the lifespan of the hardware is reduced and how this affects typical use. Since the lifetime for flash cells is typically specified as the maximum possible number of P/E cycles, we use that here as well.

4. Implementation

This section deals with our prototype implementation of the design described previously. It begins with a detailed description of the cryptographic layer between UBI and MTD and covers the steps required to encrypt all flash partitions on the N900 smartphone, managed and unmanaged[1]. This includes the root filesystem partition, user data partitions and removable media. As a final point we explain the implementation of some of our benchmark tools and test methods.

4.1. UBI cryptographic layer

As pointed out earlier, UBI is in many respects similar to the device-mapper. Consequently, we can reuse some concepts from dm-crypt when implementing our own cryptographic layer for UBI. The interface between data and cryptographic layer differs and understanding how it is implemented is crucial before we continue with our part of the implementation. Most of our code is located in a separate file, however in order to integrate it into UBI, there are a few function that must be patched. We try to keep the amount of modifications to the original UBI code to a minimum and ensure that our code is only compiled into the UBI module if this is explicitly requested by enabling the respective configuration setting in the kernel configuration.

4.1.1. Interface to the MTD subsystem

UBI uses in total six functions provided by the MTD layer to access its devices, of which only three matter to us:

- read
- write
- erase

These are the only function calls that operate on data that passes between MTD and UBI layer and thus are our main focus. Fortunately for us, all six of the related function calls are conveniently assembled in one single file and can be found in the contexts of only three different UBI functions: *ubi_io_read*, *ubi_io_write* and *do_sync_erase*. As the names already suggest, these three are responsible for reading, writing and erasing data, respectively. Based on their parameters, they calculate the address on which to operate on the MTD device and call the corresponding MTD function. This is where we hook our cryptographic layer into UBI.

[1]Whenever N900-specific points are discussed, we try to make clear that we are directly referring to this device.

Filename	Function	[A]dded/[M]odified
drivers/mtd/ubi/io.c	ubi_io_write	M
	ubi_io_write_crypted_aligned	A
	ubi_io_read	M
	ubi_io_crypt_read	A
	do_sync_erase	M
drivers/mtd/ubi/build.c	crypt_init	A
	ubi_crypt_init_mtd	A
	ubi_attach_mtd_dev	M
	ubi_detach_mtd_dev	M
	ubi_init	M
	ubi_exit	M
	ubi_mtd_param_parse	M
drivers/mtd/ubi/cdev.c	ctrl_cdev_ioctl	M
drivers/mtd/ubi/crypt.c	crypt_iv_essiv_ctr	A
	crypt_iv_essiv_dtr	A
	crypt_iv_essiv_gen	A
	ubi_crypt_async_done	A
	ubi_crypt_alloc_req	A
	ubi_set_key	A
	ubi_crypt_setup	A
	ubi_crypt_destroy	A
	do_convert	A
	ubi_crypt_convert	A
	ubi_crypt_init	A
	ubi_crypt_exit	A

Table 4.1.: Modifications and additions to UBI

4.1.2. Patching ubi_io_read and ubi_io_write

Since the modifications to *ubi_io_write* and *ubi_io_read* are slightly more complex than the ones to *do_sync_erase*, as we will see, we place our code into two separate functions called *ubi_io_write_crypted_aligned* and *ubi_io_crypt_read* and patch the invocations into the respective original functions.
The read and write functions both receive an UBI device descriptor, a pointer to a buffer[2], a PEB number, an offset and a length as their parameters. It is our mission to encrypt or decrypt the data passing between buffer and MTD read or write call, respectively. Of course, it is not quite as simple as passing the buffer as is to an encryption or decryption routine. We cannot assume that a request is aligned to block boundaries and that its length is a multiple of the block size. This is less of a concern to us for writing data because we are required to write entire PEBs, regardless of the requested buffer length and offset. Nevertheless, we have to read the existing data from the flash before we can update it and store it in a temporary buffer. This step is of course not necessary if the buffer size is equal to the PEB size. This occurs only in very rare situations however, for example when converting an existing unencrypted UBI device[3]. Since the first bytes of the block are occupied by the EC header, they are never written in combination with other data, except for the special case just mentioned.

Since we divide each PEB into sub blocks, for which individual IVs are calculated, even the smallest modification to the plaintext of each sub block result in a different ciphertext for the whole sub block, which means that we always have to work with at least as many bytes as there are in a sub block. It is also possible that a request overlaps two or more sub blocks. Both *ubi_io_crypt_read* and *ubi_io_write_crypted_aligned* take additional steps to allow the processing of unaligned I/O requests. For this, the offset is aligned to the corresponding block boundary and the requested read length is increased to a multiple of the block size, if necessary. *Ubi_io_write_crypted_aligned* then decrypts part of the temporary buffer containing the encrypted PEB data, bounded by the aligned offset and length. It then copies the data from the source buffer to the originally requested offset in the temporary buffer and re-encrypts the previously decrypted part, now also containing the new data.
Before the block can be written, it must be erased. For this we call *do_sync_erase*. We say a few more words about this function in the next section. We then encrypt the updated EC header and place it in the temporary buffer before it is written back to the flash by passing the temporary buffer to the MTD write routine.

For *ubi_io_crypt_read*, we also use a temporary buffer to hold the data to be read from the device because just like for *ubi_io_write_crypted_aligned*, we have to read more data than fits into the target buffer if the request is not aligned or crosses sub block boundaries. The temporary buffer is filled by reading from the MTD device and then decrypted, after which the requested data is copied into the target buffer.

[2]We call this buffer *source buffer* for *ubi_io_write_crypted_aligned* and *target buffer* in the context of *ubi_io_crypt_read*.
[3]We handle this situation separately and update the erase counter (EC) manually because we are erasing the block independently of the wear leveling system.

The source code for both *ubi_io_crypt_read* and *ubi_io_write_crypted_aligned* can be found in Appendix C.

4.1.3. Patching do_sync_erase

The *do_sync_erase* function is responsible for synchronously erasing PEBs. This normally restores all bits on the flash chip belonging to this block to 1. However, as we explained in Section 3.3.2, we want to make empty space unrecognizable on the encrypted device. We do so by encrypting a block of 0xFF bytes and writing it to the empty space on the flash medium, which also ensures that UBI treats these blocks as empty when scanning them. Since the erasing is performed by *do_sync_erase* anyway, all we need to do is add a small piece of code to the end of the function that takes care of writing encrypted 0xFF bytes to the previously erased block. How this overhead affects performance and flash wear is evaluated in Section 5. Obviously, we do not want to fill the block we just erased if the function is called from *ubi_io_write_crypted_aligned* when writing new data, so we add a parameter to *do_sync_erase* that controls whether 0xFF bytes need to be written or not.

4.1.4. The UBI crypto module - ubi_crypt_convert

This module represents the UBI crypto layer as we referred to it so far. It provides the encryption and decryption function *ubi_crypt_convert*, which is used by *ubi_io_crypt_read* and *ubi_io_write_crypted_aligned*. Furthermore, it manages the encryption parameters associated with a UBI device, the entirety of which we call crypto configuration. This configuration contains, amongst other things, the names of the cipher mode and associated mode of operation in use, the encryption key and pointers to various memory pools used throughout the code[4]. These parameters are configured by a setup routine that is called when a new MTD device is attached to UBI. While it is possible to pass them to the module when probing it, it is unsafe to pass the encryption key as a command line argument. An attacker could potentially read it by monitoring the running processes and their arguments. As a more secure alternative, *ubiattach* can be used for this purpose similarly to cryptsetup for dm-crypt. It passes sensitive information through an input/output control (ioctl) to the kernel which an attacker cannot easily eavesdrop on.

The crypto module also handles the communication with the Linux crypto API. For this, a cryptographic context must be created and can then be used for subsequent API calls. The parameters for this context are passed down from the setup routine and instruct the crypto API which algorithm and mode are to be used. The decision which implementation is employed is totally transparent to the crypto module and is instead chosen based on a priority value associated with each implementation. Hardware implementations generally have higher priority values than software implementations and are thus automatically used when loaded by the kernel. This can be controlled through the kernel configuration by either having them built into the kernel or building them as modules and manually probing them.

[4]Memory pools provide a number of preallocated memory blocks for use in places where statically sized blocks are allocated and freed frequently.

For decryption and encryption the crypto API expects a scatter/gather list instead of a pointer to a buffer. A scatter/gather list contains pointers to page-sized contiguous memory regions which are required for Direct Memory Access (DMA) operations. This type of memory access is performed by the DMA controller without involving the CPU. The DMA controller receives all information necessary for the task of copying data between a peripheral device and main memory and executes it in the background. This reduces computational overhead and thereby improves the throughput for transferring data to and from a device [30]. With data already prepared for scatter/gather I/O, device drivers for cryptographic co-processors registered with the crypto API can simply use the data for DMA operations without having to perform the conversion themselves. Since we receive all data as buffers from the UBI layers, this task falls to us. One of the memory pools created when setting up the module is responsible for allocating pages necessary for this operation. The actual scatter/gather list generation is assisted by macros and functions provided by the kernel for this purpose, as this is a very common operation. Prior to it, the buffer is split into pieces equivalent to the minimum I/O unit supported by the underlying MTD device for IV generation. The method used for this is borrowed from dm-crypt, which needs to perform the same task and provides support for a variety of IV modes.

We use the asynchronous block cipher interface for its support of hardware accelerated encryption and decryption. Aside from that, one of the major differences to the synchronous block cipher interface is that it provides batch processing by queueing requests until the current job is complete and notifies the initiator of the request upon its completion, regardless of whether hardware acceleration is in use or not [31]. Instead of using a busy-waiting loop, we can use completion events which put the caller to sleep and reschedule it when a certain condition is met, a feature that was added to the Linux kernel in 2.4.7 [32]. We use this in particular to wait for an encryption or decryption request to finish before the result can be handed back to the UBI I/O routines. Even though our code does not support parallelization at the time of writing, the use of asynchronous block ciphers also provides a base for adding this optimization later.

The source code for *ubi_crypt_convert* is included in Appendix C.

4.1.5. Attaching to and initializing MTD devices

As discussed in Section 3.3.4, we rely on user-provided information to distinguish between encrypted and unencrypted devices. There are two ways to do so, the first being the UBI module parameters. When probing the UBI module, an MTD device can be passed as parameter, resulting in this device being attached to UBI during the module initialization. The parameter can either be passed using the modprobe command or on the kernel command line if UBI is built into the kernel. By extending the device name with a flag we can indicate which action to take for this MTD device. Device name and flag are separated by a colon. The second method is using *ubiattach*, which can also be used to attach MTD devices to UBI after UBI has already been loaded. The communication between *ubiattach* and UBI takes place using ioctl commands, which we extend with an additional command for attaching encrypted devices. The advantage of the second method is that it also hampers potential eavesdroppers.

Besides attaching to MTD devices, it is also important to be able to encrypt previously unencrypted devices and to initialize empty devices for encryption. Incidentally, these two tasks are equivalent in our design. Encrypting an existing device requires every block on it to be read, encrypted and rewritten to the device. Since empty flash memory contains 0xFF bytes, it can be treated in exactly the same manner. Obviously, each block must be erased before it can be rewritten. The unnecessary step of rewriting it with encrypted 0xFF bytes, as described in 4.1.3 must be skipped in this case, saving one programming cycle. The crypto initialization routine *ubi_crypt_init* is hooked into *ubi_attach_mtd_dev*, the UBI function that takes care of attaching to MTD devices so that the device is readily available after initialization. Another important factor of using this method to convert existing UBI devices is that erase counters are preserved.

Alternatively, it is also possible to flash a pre-encrypted UBI image onto an MTD device. If this device contained UBI volumes prior to this, it causes erase counters to be replaced and the UBI wear leveling algorithm cannot take into account how often blocks have been erased in the past. This can be prevented by creating the image from a backup of the old content of the MTD device.

4.1.6. Build.c and cdev.c changes

Since encryption needs to be available as soon as an MTD device gets attached to UBI, we need to patch the functions involved in attaching and detaching UBI devices in `build.c`. There are two methods of adding a device to UBI: passing it as parameter when probing the UBI module and by invoking *ubiattach*. The communication between this user space program and the kernel module is done using ioctl commands. The interface for this can be found in `cdev.c`, where we modify the attach command to support additional parameters which control whether the device being attached is treated as encrypted. They also allow formatting and conversion of an unencrypted device to an encrypted one.

4.1.7. Key management

We described our key management design in Section 3.3.3. Essentially, this is realized together with the *ubiattach* support for attaching MTD devices. Since attaching an encrypted MTD device requires knowledge of the encryption key and parameters, it is only sensible to pass this information in the same step. Because we obviously do not want to store the passphrase in plain text on the device, the user has to somehow enter it during the device bootup. This can only be achieved by adding an init script that calls *ubiattach*. Normally, the N900 does not use an init script so we need to find a place where to place an initrd that contains the init script and all required binaries. The fifth partition is already named aptly for this as we mentioned in Section 3.1.3 and provides sufficient room for a small initrd.

To reduce the effort of repeatedly entering all these details for our measurements, we store them statically in the kernel module for our prototype.

4.1.8. Boot process

While information about the boot process of the OMAP 3430 is available from Texas Instruments, the boot loader on the N900 is a proprietary piece of software called Nokia Loader (NOLO). It supports booting from various devices, such as the MMC card and USB. This is relevant if the kernel and initrd are to be moved to an external medium. Doing so increases security slightly because it both prevents modifications to either by a malicious software running on the device and furthermore prevents any conclusions about the filesystem and encryption used. This information can potentially be obtained from the init script contained in the initrd, depending on the configuration. Indeed, the reduced effort of not having to enter this information manually during bootup could be regarded as a counterbalance for the additional effort of keeping kernel and initrd separate. By default, there is no initrd on the N900, however a MTD partition exists which can be used to hold a 2 MB initrd. We create an init script that takes care of detecting and mounting the root device. Our prototype implementation has the key required for decrypting the device built-in, therefore we do not need to query it from the user here, however this could easily be added. One common example for this is a GNU Privacy Guard (GPG) encrypted file containing the cryptographic key material, stored on a separate partition or external medium and decrypted at this stage after querying the user for the passphrase. We place the init script together with the kernel modules and programs required to mount the root partition in an initrd and instruct the kernel to load it during bootup. One main advantage of this is that we can build *omap-aes* as loadable module, which allows us to load it early enough in the boot process for it to be available when the root partition is mounted. Otherwise, *aes_generic* is used by default, which cannot be changed once mounting has finished.

4.2. Encrypting the N900 filesystems

The previous sections should give a good overview about what is necessary and what is optional for encrypting the root filesystem on the N900. In the standard configuration, there are three additional partitions on the internal MMC card that can be encrypted[5]. These are

- **/dev/mmcblk0p1**
  Mounted on /home/user/MyDocs, vfat
- **/dev/mmcblk0p2**
  Mounted on /home/user, ext3
- **/dev/mmcblk0p3**
  Used as swap

Since all of them are located on a managed flash device, they can easily be encrypted using dm-crypt. Because dm-crypt relies on the crypto API as well, it can equally benefit from the cryptographic co-processor. The necessary tools such as cryptsetup and the dm-crypt kernel module are readily available from the Maemo package repository. Since the exact process of encrypting partitions on the N900 using dm-crypt does not differ from regular desktop systems, we do not cover it in detail here.

[5]Not counting those on an optionally inserted external MMC card

4.3. Benchmark tools

In this section we provide a few more details about the self-written tools used for measurement and evaluation. This makes it easier to understand the logic behind the numbers presented in the evaluation section.

4.3.1. Kernel module

Because our code is exclusively accessible from kernel mode, we need to move some of our benchmarking to the same level. We need to measure the throughput of both the raw encryption and decryption routines for a direct comparison between hardware and software implementation and we want to measure the performance of individual UBI crypto layer functions.

4.3.2. Scripts and tools

- **runbench**
  To manage benchmarks that have to be run repeatedly and for different scenarios, for example with and without hardware crypto acceleration, we use a simple Bourne-Again SHell (bash) script named runbench. It takes care of starting the relevant processes and stores log files for later evaluation. Furthermore, it loads the oprofile kernel module and starts the daemon to log profiling information.
- **watchproc**
  To measure the overall load on our device while performing benchmarks, we continuously monitor the stat entry of the proc filesystem. Reading this file provides values for the time the system spent in user mode, system mode and idle mode, amongst others.
- **oprofile**
  As explained in Section 2.1.5, oprofile collects profiling information of all running user and kernel processes. It provides two tools for analyzing the data subsequently: opreport and opannotate. The former can generate detailed reports and callgraphs for all or specific binaries it collected data for, the latter produces annotated source code for binaries compiled with debugging symbols. The annotations include the number of samples collected for each line of code or optionally for each assembly instruction, and the percentage in relation to the number of samples collected overall.

4.4. Portability

One of our stated goals is the portability to other devices than the Nokia N900. We designed a system that can easily be ported to other hardware that is supported by the Linux kernel and provides an unmanaged flash device. Since UBI supports other filesystems by means of *gluebi* and emulated block devices, there is no requirement to use UBIFS. Of course, UBIFS and UBI were designed to complement each other and work with no intermediate layers, which is likely to result in better performance and stability than other approaches. Nevertheless, it may be desirable to use a different filesystem, depending on the conditions. Since other flash filesystems such as YAFFS and JFFS2 use the same interface to access the underlying hardware as UBI, it is possible to port the changes we made to UBI to these filesystems. Most of our implementation is generic enough and does not depend on UBI specifics so that this process should be fairly painless. This is mainly thanks to the fact that we operate very close to the MTD layer. It should be possible to integrate a slightly modified *ubi_crypt_convert* into both JFFS2 and YAFFS, however some additional changes are likely to be required to either filesystem.

5. Measurements and evaluation

After having proven that our design works in practice, it is now time to take a look at how well it performs. In the following we present our measurement methodology and discuss the results from our measurements as well as analyze and evaluate them. This includes benchmarks of the software and hardware AES implementation, filesystem and power consumption benchmarks as well as flash endurance related findings. Measurements are done using our prototype implementation on the N900, for which we use iozone to obtain detailed reports and then use bonnie++ to confirm some of the key results. To give us an overview of where CPU cycles are spent most, oprofile runs in the background for some of the benchmarks. We start with raw throughput and filesystem performance measurements, continue with power consumption and finish the measurement part with flash wear. Some of the benchmarks run for many hours and as explained in 5.1.1, we rely on a fairly small sample size for these.

5.1. Methods and tools

After having determined what we want to measure in Section 3.4.1, we now take a closer look at measurement methodology. Since we are measuring on a multi-tasking capable system on which daemons and other processes are running in the background, it is unavoidable that we have a certain level of interference. In other words, we expect a large amount of variance, which can only be compensated for by a large sample size. Unfortunately, due to time constraints and the duration of some of our benchmarks, our sample sizes are very small. Since this prevents us from drawing any conclusions about the sampling distribution these samples are drawn from, we will include error bars depicting the Standard Deviation (SD) instead of the Standard Error of the Mean (SEM) in our barplots.

We use the N900 smartphone for all of these measurements, as it is a very good reference platform. In addition to the tools that come with the Maemo distribution, a Debian change root (chroot) package exists that provides access to the large set of packages compiled for ARM CPUs available from Debian and which we can utilize for our purposes. Among these is the well known bonnie++ benchmark suite aimed at testing hard drive and filesystem performance [28]. It allows us to measure the effects on filesystem performance when using our UBI cryptographic layer and even reports the average CPU load observed during the benchmarking process. It is important to note however that this is an average over the system load and therefore includes the load from unrelated processes. As a consequence, this value does not allow any definite conclusion about the speed of the individual components of our code such as encryption and decryption routines. To benchmark these, we use our previously mentioned kernel module which implements various test cases for this purpose. As a more accurate way of determining this module's CPU load we use a small program that continuously monitors the CPU load percentage of a specified process and a Perl script to evaluate its output. For measuring responsiveness, we use a less sophisticated approach: application startup time is taken using a stopwatch and installations are timed using the Linux time command[1].

To determine how much cryptography affects the maximum uptime achievable with a fully charged battery, we take advantage of the fact that the power management automatically shuts down the device once the remaining battery charge drops below a certain level. Unfortunately, it is not trivial to create a simulation setup that models typical user behavior accurately, therefore we limit ourselves to benchmarks and ignore influences not directly related to filesystem encryption in the scope of this thesis. Two important examples for these influences are network related functions and the Liquid Crystal Display (LCD), both of which can be expected to be comparatively power hungry.
We measure the number of benchmarks cycles that can be completed before the battery is completely drained and continuously record the power level as reported by HAL during the execution.
To measure the effect on flash endurance, we want to measure the number of P/E cycles performed and compare the numbers for encrypted versus unencrypted UBI devices. We can take advantage of the fact that UBI keeps erase counters for each PEB. This part of the wear leveling system which ensures that I/O load is spread across the whole chip. PEBs are typically erased when data is written to the medium, when the wear leveling system relocates data away from blocks with high erase counters or when scrubbing[2] takes place. Since almost every block erase is followed by a block write, we use the erase counter to determine the number of P/E cycles.
To get comparative results for both the software and hardware implementation of AES in CBC mode, we run most tests with both implementations while using the same parameters.

5.1.1. Test cases

Bonnie++ covers multiple real-world scenarios such as linear and random file access, file deletion and creation. These are sufficient for our needs and provide meaningful answers to our question about the effect on filesystem performance. As mentioned above, bonnie++ also reports the average CPU load during the duration of the benchmark, allowing us to compare overall system loads of accessing an encrypted versus an unencrypted filesystem. As a comparison between unmanaged and managed flash, we run these tests both on an UBIFS volume and an MMC partition, the latter formatted as ext3.

To measure throughput and load caused by the raw encryption process, our benchmark module repeatedly encrypts blocks of increasing size from 16 bytes to 256 KB using the same cryptographic context (CTX)[3] for a predefined length of time and prints the amount of data processed during that time afterwards. Since only support for AES in CBC mode is implemented by our prototype, we limit our tests to this specific configuration.

[1]The time command displays the time elapsed between the invocation and termination of a command, the CPU time used by it and the system CPU time.

[2]Scrubbing is the process of moving data from PEBs with bit-flips to other blocks. Bit-flips can be corrected by ECC checksums but they can accumulate over time [29].

[3]Consists mainly of key, algorithm, and cipher mode.

We are also interested in determining how the encryption of individual or all partitions affects daily use on the N900. Since the definition of daily use depends on the respective user, we chose a few sample scenarios that we consider typical for smartphone usage. As test cases for measuring execution time, we choose

- Application startup time
- Installing a new application
- Cold-booting the OS

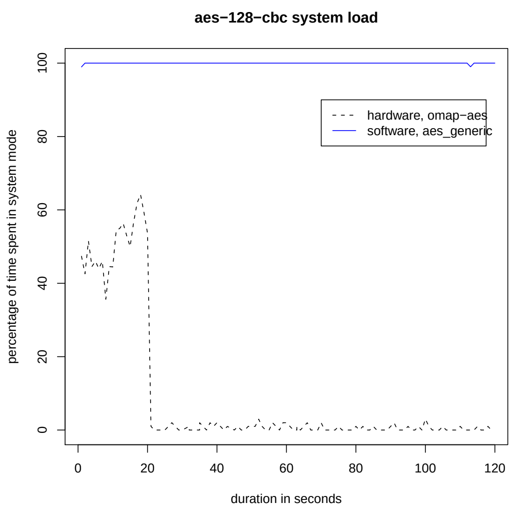
We already mentioned the need for two test cases for measuring power consumption and maximum uptime. Simulating no user interaction is obviously very simple. To simulate typical smartphone usage we can reuse the tasks used for measuring execution time and execute them repeatedly over the time until the battery is completely drained.

To calculate the number of P/E cycles, we mount a newly created UBI device and run iozone on the UBIFS volume, after which we unmount the volume and read out the erase counter for each PEB. We do this for both an encrypted and an unencrypted volume and then compare the results.

For all scenarios listed above, we take multiple samples to calculate the mean over the respective results. This is especially important for tests where we expect high variance. However, some of our benchmarks run for a very long time or are relatively complex to automate, therefore we have to resort to using a small number of samples due to time constraints. For instance, iozone can run for hours under certain conditions, therefore we limit our sample size in some cases and accept the negative effects this has on the statistical significance of our results.

5.2. Performance: throughput

Our first benchmark measures the raw throughput achievable on the N900 using the *omap-aes* hardware accelerated AES driver and the *aes_generic* software implementation of AES. We do this by counting the number of fixed-size blocks that can be encrypted in a specified time frame. This ensures that we only measure the actual encryption process and exclude buffer allocations and initializations from the results, barring those performed by the driver. For the sake of completeness we also measure the number of blocks that can be decrypted in the same time frame. Figure 5.2 shows our results for block sizes between 16 bytes and 256 KB over a 5 seconds interval. Note that the x-axis, representing the block size in KB, uses a log scale. As it can be seen, the software implementation appears to be faster for very small block sizes but quickly loses its advantage and is surpassed by the hardware implementation at block sizes larger than 2 KB. This is a good time to recall that the memory page size for the ARM architecture equals 4 KB, which coincides with the block size at which the growth rate for the graph representing the hardware implementation begins to subside. Since data between *omap-aes* driver and the underlying hardware is passed using DMA, it needs to be split up into chunks of individual pages. For requests that are smaller than one page, the communication between driver and hardware seems to cause enough overhead to slow the whole encryption process down by a measurable degree. In total, we reach a maximum of about 8 MB/s using the *omap-aes* driver and approximately 6.5 MB/s using the *aes_generic*



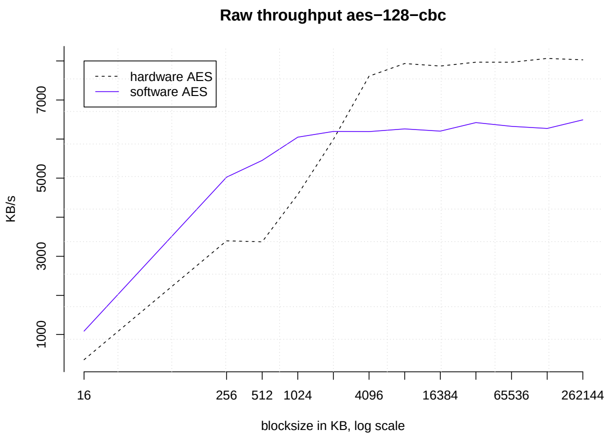


Figure 5.1.: system load, hardware vs. software

Figure 5.2.: throughput, hardware vs. software

implementation. The values are roughly equivalent for encryption and decryption, which is not surprising, as the inverse cipher of AES is an inversion of the transformations done by the cipher, implemented in reverse [33]. It remains to be seen if we can observe a similar trend during our filesystem benchmarks.

Apart from the speed measurements, we are also interested in the system load during our benchmarks. For this purpose we collected samples from /proc/stat in 1 second intervals. Since these values represent the overall system load, they of course include interference from other code running alongside our module. Nevertheless, this should give us a rough estimate of how much additional load is caused by our benchmark. To reduce said interference, we run our benchmark multiple times and use the mean value of all results combined. Naturally, we expect a measurably lower load when using the hardware AES implementation, compared to the software implementation. Figure 5.1 shows the system load during our throughput experiments, with the time of the experiment on the x-axis and the system load on the y-axis. It is important to note here that there is a correlation between time and block size. Over the course of the experiment, the block size is increased every 5 seconds. The system load for the aes_generic implementation is almost constantly 100%, while system load for the *omap-aes* module stays below 5% most of the time. There is a large spike in the *omap-aes* graph between 0 and 20 seconds, which coincides with the interval during which block sizes of 16 and 256 bytes are tested. This is not entirely surprising, if you consider that in case of smaller block sizes, the CPU needs to spend more time of the 5 second interval calling crypto API functions and managing data than for larger block sizes. For example, the number of individually encrypted blocks is roughly 74% higher for 16 byte blocks than for 512 byte blocks. All in all, our expectations concerning system load were met.
The most important conclusion we can draw from this result is that using the OMAP 3430 AES co-processor on the N900 is both faster and more efficient in terms of overall system load for block sizes of 2048 bytes and above than running the software implementation on the ARM CPU.

5.3. Performance: iozone

Let us now take a big leap and look at how much of this advantage remains when running iozone on an UBIFS volume on an encrypted UBI device. For this, we move the root filesystem of the N900 to the integrated MMC so that we can use an empty UBI volume for benchmarks. We then erase the respective MTD partition, create an empty, encrypted UBI volume on it, mount it and run iozone. Afterwards, we repeat the same process but insert the *omap-aes* beforehand, which ensures that all aes-cbc operations are handled by the cryptographic co-processor. Figure 5.4a and Figure 5.4b were generated using a script included with iozone. The former shows the write performance using *aes_generic* while the latter shows the measured read performance. As mentioned before, iozone runs several tests with increasing block sizes, which are shown on the x-axis, over different file sizes, which are shown on the y-axis. The resulting performance is displayed as a grid, measured in KB/s. One important thing to note before we start analyzing the plots is the fact that iozone does not test block sizes below 64 KB for files above 16 MB by default, an extra feature we do not make use of. Naturally, the block size cannot exceed the file size. To create a surface without holes, these undefined values are interpolated in the plot.
When looking at the data represented by Figure 5.4a and 5.4b, the first thing we notice is the different relation between block size and write/read speed, compared to our previous throughput measurements. The same is true for Figure 5.5a and 5.5b, which represent the iozone runs using *omap-aes*. There are some striking differences between the results from our throughput benchmarks and the ones from our iozone benchmarks. First, the relation between block size and speed differs. As an example, using a block size of 64 KB yields a maximum read rate of about 340 MB/s while a block size of 16 MB yields only about 157 MB/s, both using *aes_generic*. These values were measured for file sizes of 64 KB and 16 MB, respectively.
Second, the measured speeds greatly exceed the maximum rate at which we previously were able to encrypt and decrypt during our throughput benchmark using either method. This second observation is relatively easy to explain. Since iozone was not explicitly instructed to flush the cache while running the benchmark, we end up measuring the effectiveness of the file system cache, or more precisely the page cache. As explained in Section 2.1.4, all reads and writes from user space go through the VFS, which caches recently used data. This causes I/O rates which go beyond what the underlying device is capable of, as we can also observe in our plots. Eventually the cache fills up though, which is visible in the measurements as a sharp drop in the read and write rates for file sizes of 128 MB, best visible in Figure 5.7a and 5.7b. These show a selection of file and block size combinations for the same benchmarks as Figure 5.4a and 5.4b, respectively. We also notice a high amount of variance in our results for some combinations, which seems particularly pronounced for write operations. Since the cached I/O rates are more limited by overall system load and memory utilization than by the actual device I/O, it is likely that our results are affected by the fact that CPU and RAM are shared resources, possibly in use by other processes. This effect is most evident for write access, which generally requires more work than reading due our design decision that requires additional steps when updating a PEB.
Even though these results do not directly reflect the performance of our implementation, they are valuable to determine the actual impact of our implementation on typical everyday usage as it is simulated by the benchmark. There are obvious similarities in the results for

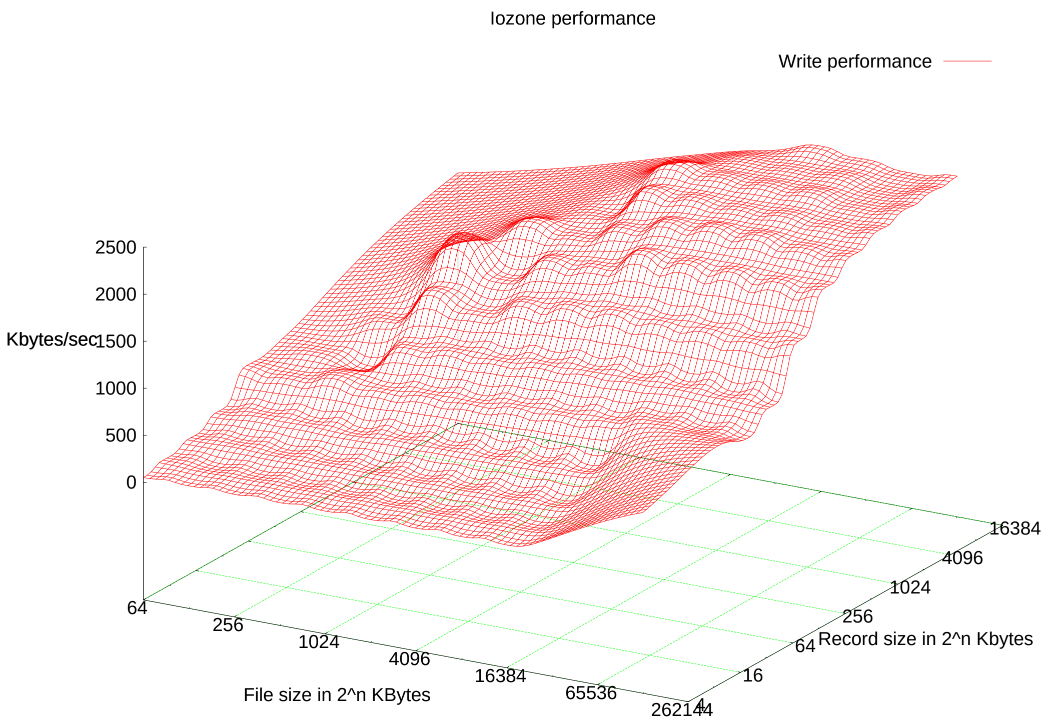


Figure 5.3.: Synchronized write performance, *aes_generic*

plaintext and for encrypted UBI devices, both using hardware accelerated and software encryption. All iozone plots exhibit similar trends of performance degradation with growing file sizes, although this is more pronounced for write tests than for read tests.
Earlier, we pointed out the different relation between block size and speed when comparing throughput benchmarks to iozone benchmarks. To investigate this further, we need to take a look at the actual performance that is achieved when writing synchronously to the flash filesystem, leaving the cache out of the calculation. This is supported by iozone by means of two switches. The *-l* switch instructs iozone to use direct I/O, which unfortunately is not supported by UBIFS, leaving us the *-o*, which forces iozone to open files in synchronous mode and thus includes the time required to flush the page cache in its calculations. It ensures that iozone waits for the cache to be written out to the UBI device before regarding the operation as complete. This does not have any effect on the read performance, as there is no way to disable the cache for this, excepting direct I/O, which we cannot use. We will get back to this later and present a different solution for judging the actual read performance of an encrypted UBI device.
Figure 5.3 shows the results from running iozone with synchronized writes using *aes_generic*. As we can see, the difference to our previous measurements spans up to one order of magnitude for some file size/block size combinations. While read speeds are unaffected due to the continued use of caching, write speeds are now in the range of a 100 KB/s to 2.5 MB/s, compared to a maximum of 200 MB/s for our unsynchronized experiments. This is closer to our previous throughput measurements but significantly slower. Interestingly, there is a

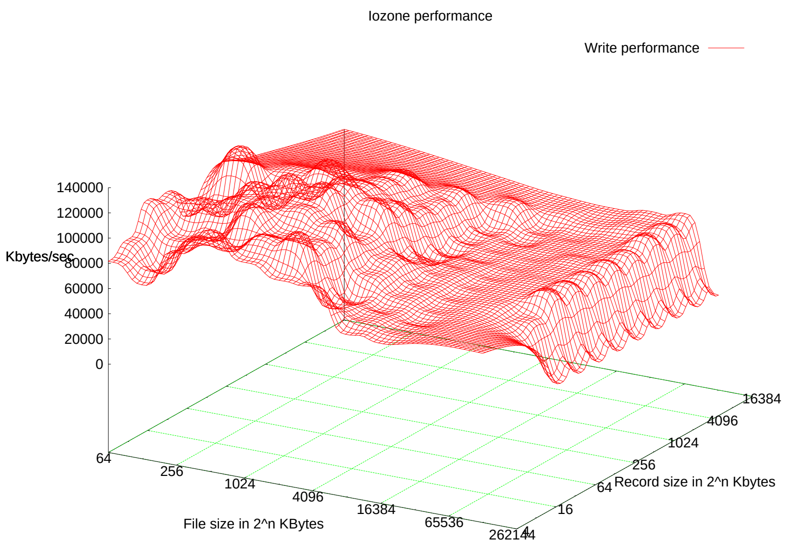
similar relation between block size and speed now, although iozone operates at block sizes starting at 4 KB while our throughput benchmarks start at 16 bytes. We also notice that a block size of 4 KB yields notably lower speeds in comparison. Since 4 KB sticks out as a turning point for our throughput measurements, it is interesting to see that this is not the case for the iozone benchmarks. The reason for this is the large amount of overhead from data always being written in PEB-sized blocks. This corresponds to 128 KB on the N900, resulting in an overhead of 124 KB for each 4 KB block. Additionally, these blocks also need to be erased each time before they can be written, causing further delay. It should be clear that for this reason, write requests for small block sizes are considerably slower than large requests, which is also very obvious in Figure 5.3.

5.4. Performance: bonnie++

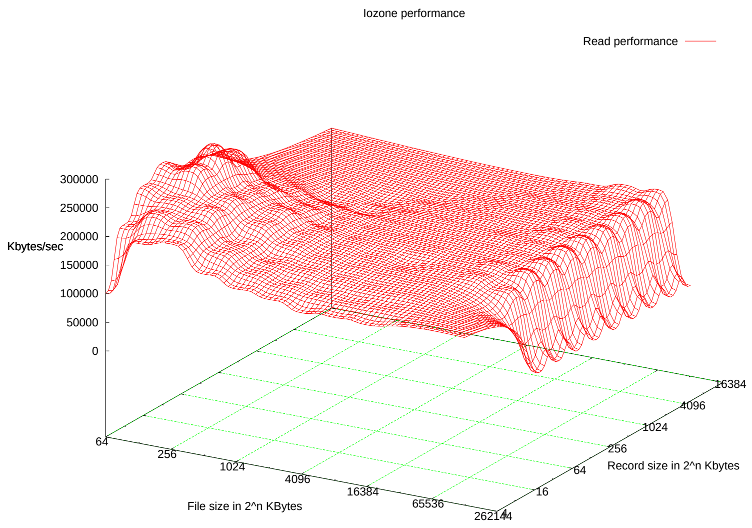
We want to confirm our latest results using a second file system benchmark, for which we choose bonnie++. Like iozone, it has an option to enable synchronized I/O for the measurements, which we make use of. In contrast to iozone, bonnie++ typically uses only one file size throughout the test run, which is dependent on the amount of RAM. The default size is size of physical RAM times two, which is larger than our UBI volume, so we have to reduce it to fit the device. For the block read/write tests, the block size is 8 KB by default. In addition to writing blocks, bonnie++ also measures the performance when writing single bytes. This is expected to be much slower than 8 KB blocks due to the large overhead from updating entire 128 KB blocks for every single byte. Figure 5.8 shows the benchmark results in KB/s on the y-axis, using a log scale. The x-axis shows the different bonnie++ I/O tests [34] used in our benchmark, which consist of the following:

- **Chr_Out**: Single byte output
- **Blk_Out**: Block output
- **ReWr_Out**: Reads blocks from the file, modifies them and writes them back
- **Chr_In**: Reads individual bytes
- **Blk_In**: Reads blocks

The results show similar properties as our iozone reports, however we notice that bonnie++ reports faster write rates despite the comparatively small block size. We re-run the bonnie++ benchmark using a larger block size of 1 MB, which causes insignificantly faster rates. Looking at the source code of both benchmarks reveals that iozone opens files with the O_SYNC flag for synchronized writes while bonnie++ calls fsync after writing all blocks and includes the required time in its results. The major difference between the two approaches is that O_SYNC causes write calls to block until all data has been written to disk. The effect is similar to using O_DIRECT for direct I/O but as mentioned earlier, only writes are affected. The approach employed by bonnie++ uses cached writes and moves the synchronization step between cache and disk after writing has finished. Obviously, this causes significantly less overhead for small block sizes because the synchronization is not invoked for each individual block but instead as a commit for all modifications at the end. Iozone supports this mode of operation as well with the *-e* command line switch.

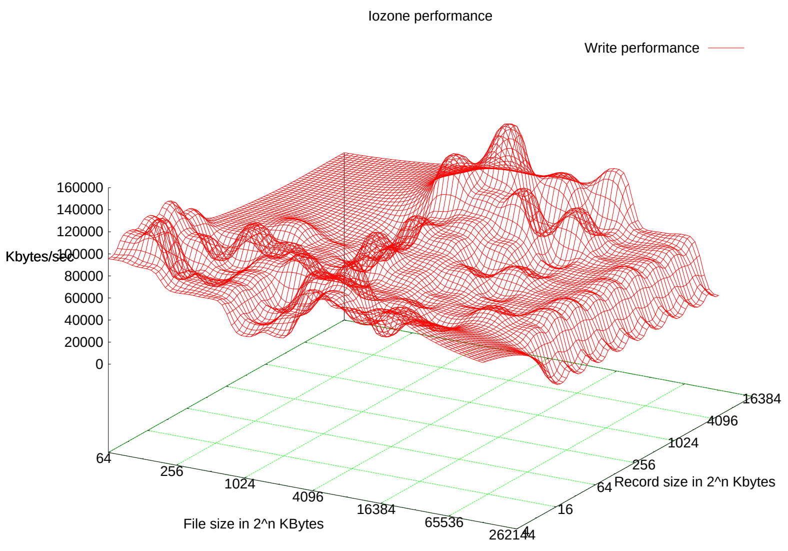


(a) Write performance

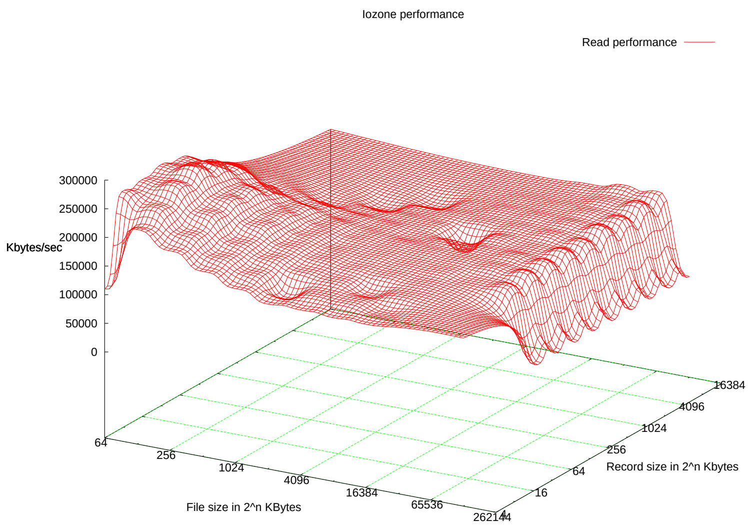


(b) Read performance

Figure 5.4.: Iozone measurements, no sync or flush, *aes_generic*

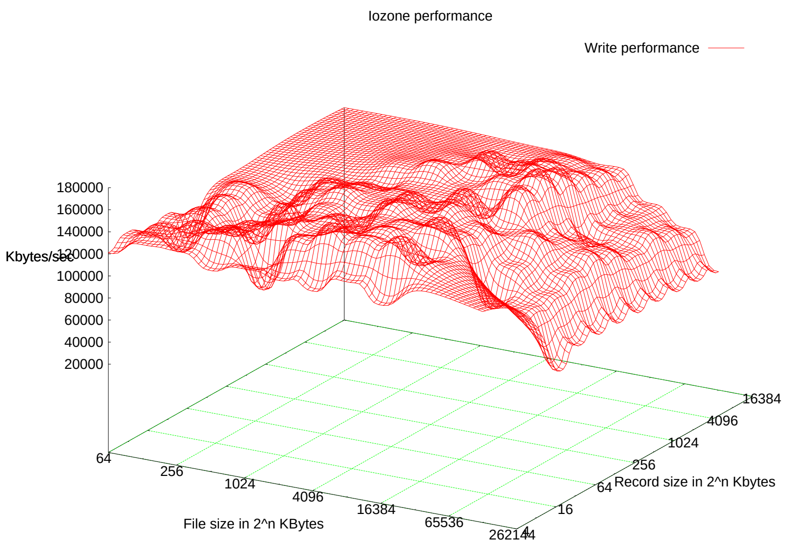


(a) Write performance, *omap-aes*

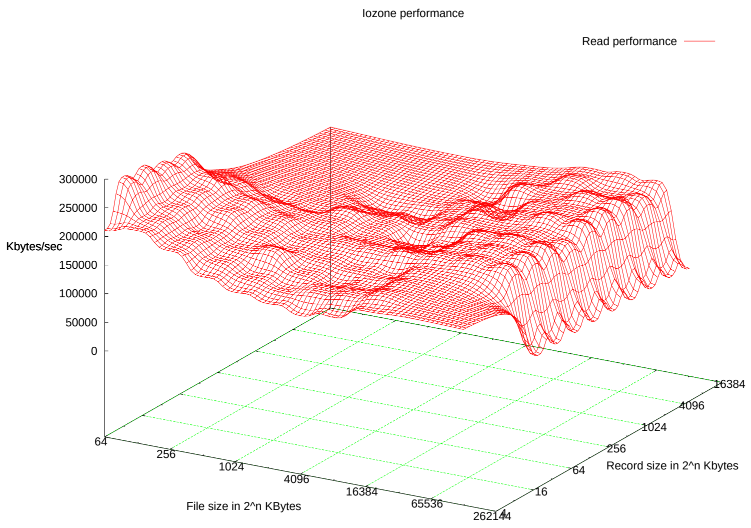


(b) Read performance, *omap-aes*

Figure 5.5.: Iozone, no sync or flush, *omap-aes*

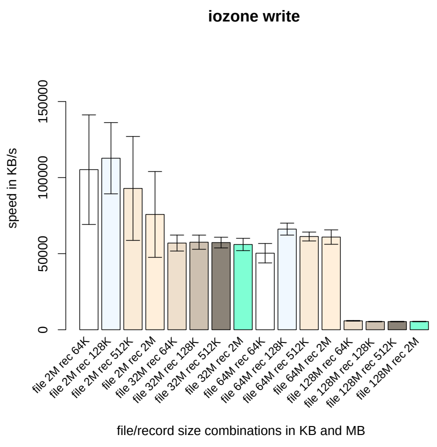


(a) Write performance, plain text

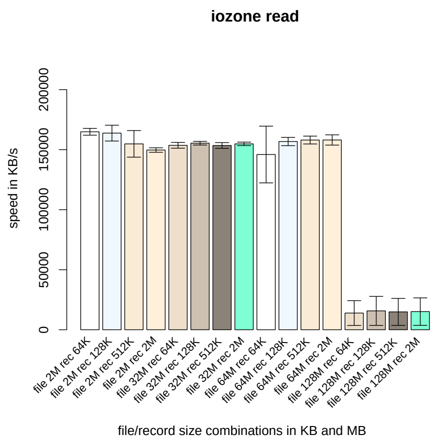


(b) Read performance, plain text

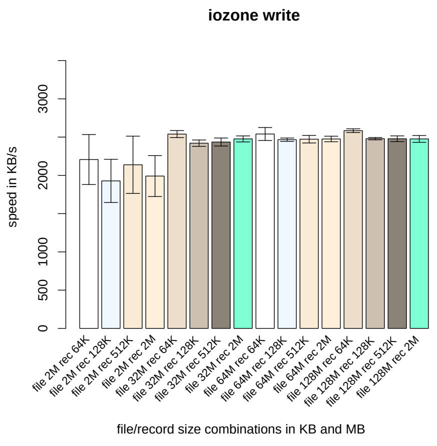
Figure 5.6.: Iozone, no sync or flush, plaintext



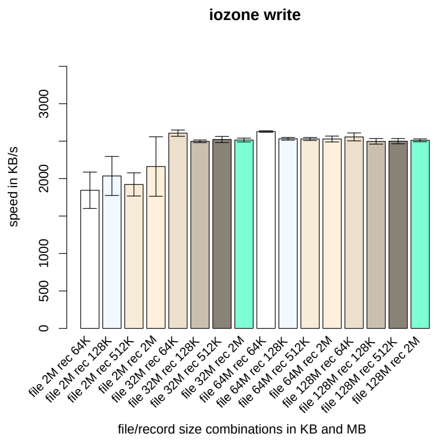
(a) Write performance, *aes_generic*, no flushing



(b) Read performance, *aes_generic*



(c) Write performance, *aes_generic*, flushing



(d) Write performance, *omap-aes*, flushing

Figure 5.7.: Iozone, multiple measurements

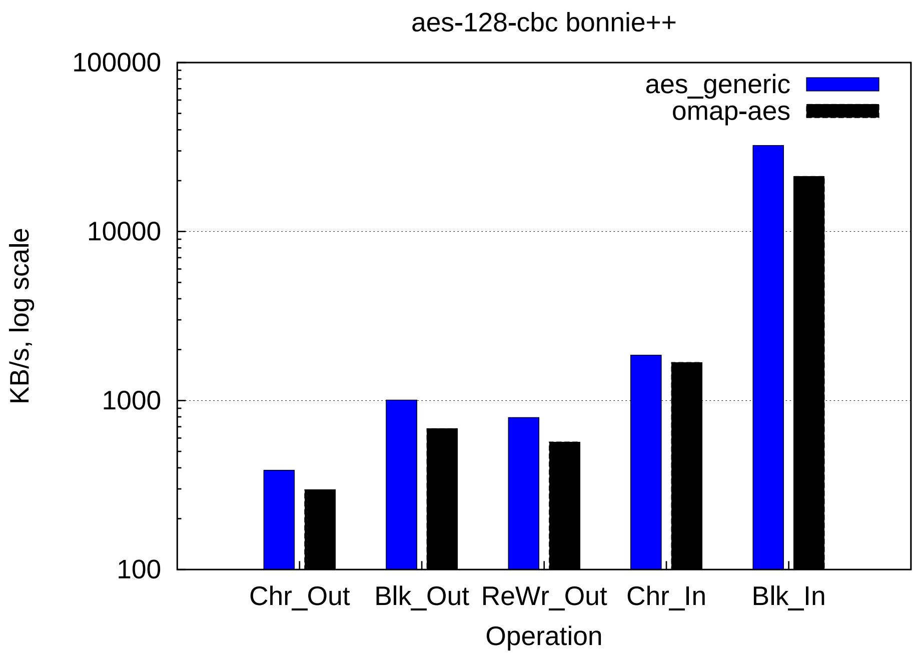


Figure 5.8.: bonnie++

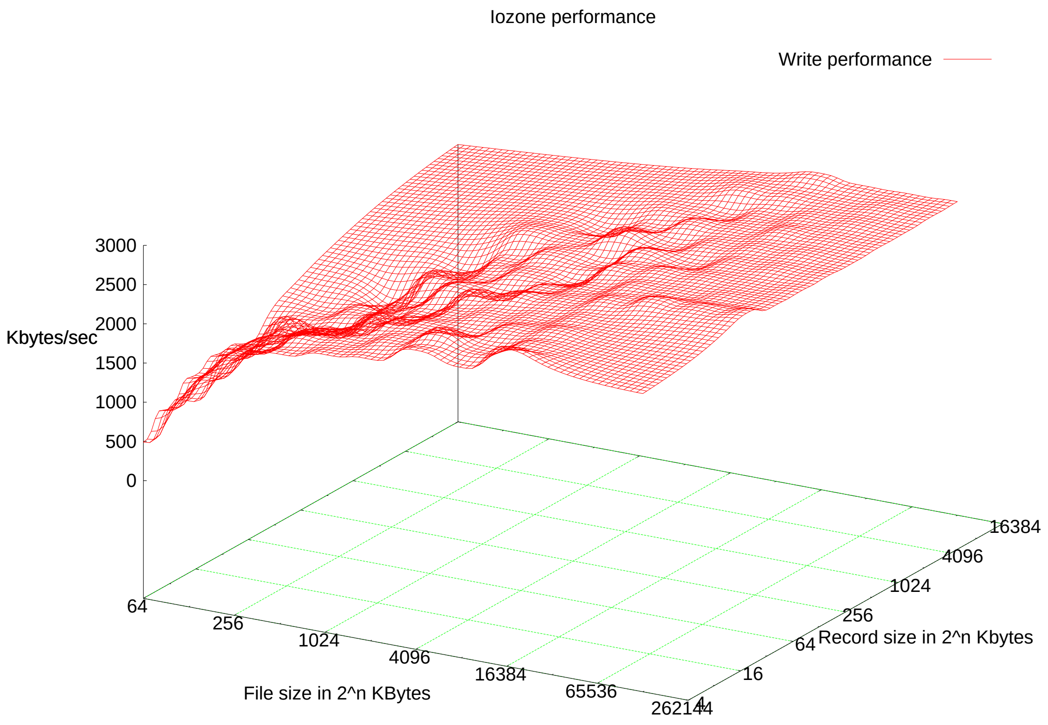


Figure 5.9.: Iozone, write, *aes_generic*, with flushing, 8 samples

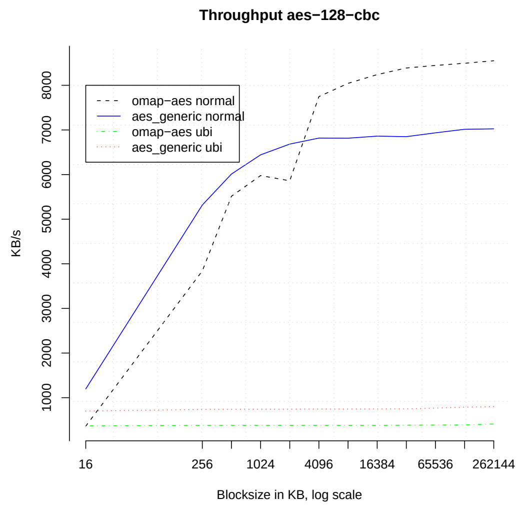


Figure 5.10.: Throughput, *ubi_crypt_convert*

This prompts us to repeat our iozone measurements with this option enabled and O_SYNC disabled for comparison. Figure 5.9 shows obvious differences for combinations of files sizes above 1024 KB and small block of 64 bytes or less, which now yield rates of 2 MB/s or more. The observed maximum is now roughly 2.5 MB/s. Figure 5.7c shows the same benchmark as Figure 5.9 as barplot, which makes it easier to make out individual values and shows information about the SD of our measurements. Apparently, the SD is particularly large for small file sizes such as 2 MB, for which we have no immediate explanation. Figure 5.7d shows the same benchmark, this time using *omap-aes*. There is no notable difference to the results from using *aes_generic*.

5.5. Performance: ubi_crypt_convert

After having confirmed our iozone results using bonnie++, we can now be quite confident that we are looking at sensible numbers. This brings us back to our question of why we see a discrepancy in speed when comparing filesystem to throughput benchmarks. Our throughput benchmark only measures the speed of the actual encryption and decryption routines from *omap-aes* and *aes_generic* while the filesystem benchmark also accounts for the different layers between a I/O request and the actual cryptographic conversion. It is likely that there is room for optimizations but before we can look into that, we need to pinpoint the bottlenecks. We must not forget that we are dealing with a proof-of-concept implementation that was in no way optimized for speed, so it is reasonable to expect the biggest issues in our code. The largest piece of work is performed by *ubi_crypt_convert*, which splits a buffer, populates a scatter/gather list and passes it to the crypto API functions. This makes it comparable to the throughput benchmark function. Let us take a look at how they compare when given the same task of encrypting as much data as possible in a time frame of 5 seconds. For this, we can simply add *ubi_crypt_convert* to our benchmark module and rerun the test we used for our previous throughput benchmark. Figure 5.10 shows the results from this endeavor in

direct comparison to direct calls to the crypto API functions and as we can see, there are noticeable differences. The most prominent one is that *ubi_crypt_convert* is considerably slower than its counterpart, no matter if hardware acceleration is used or not. In addition to that, there is no apparent relation between block size and speed. This is not entirely unexpected, as the filesystem benchmarks did not display an overly pronounced relation between these two parameters either, compared to the throughput benchmark. It is now clear that we have found a large bottleneck, which requires some closer investigation.
As we said earlier, oprofile offers itself as a tool for this task. We start the oprofile daemon and rerun our last benchmark. For analyzing the collected profiling data, we use opreport and opannotate, which we described in Section 4.3.2. In all cases, opreport returns less than 10% of the collected samples being collected in *ubi_crypt_convert*, which can be seen in Appendix D.1. Most samples are collected in *aes_generic* and *omap-aes*. Why *aes_generic* appears even when *omap-aes* is in use may not be directly obvious. *Omap-aes* provides only two AES related routines: AES-CBC and AES-ECB, however we require AES without block chaining for the ESSIV generation process. For this reason, we still depend on *aes_generic* and can only offload block encryption to the co-processor. The annotated source code for *ubi_crypt_convert*, included in Appendix D.1, allows us to confirm that there is no notable difference in used CPU cycles in combination with either AES module. What is causing the slowdown when using *ubi_crypt_convert* then? Let us take a closer look at *ubi_crypt_convert* and determine what happens prior to the data is being passed to the AES engine. As explained in Section 3.3.1, we split data on a PEB into small logical sub block for the IV generation to reduce overhead. This is essentially the main task for *ubi_crypt_convert*: it splits the buffer passed in the source buffer into logical blocks, generates IVs, encrypts the blocks one by one and stores the result in the destination buffer. The logical block size is equivalent to the smallest data block UBI operates on, which in turn depends on the smallest accessible data unit available for writing on the underlying flash device. On the N900, this is 512 bytes. This leads to an interesting situation: independent of the size of the buffer passed to *ubi_crypt_convert*, AES always operates on blocks with a size equivalent to this sub block size. Smaller buffers are rejected at the start of *ubi_crypt_convert*. This means that the different buffer sizes passed by our benchmark make no difference at all, which is also reflected by the related graphs in Figure 5.10. In order to be able to run the exact same tests with the ubi conversion function, we reduce the sub block size to 16 bytes, so during all tests, AES operates on blocks of this size. This is a very important detail because we earlier observed that using small block sizes below 2048 bytes is considerably slower than using larger blocks. We also know that 2048 bytes is close to the turning point where *omap-aes* becomes faster than *aes_generic*, which means that *ubi_crypt_convert* needs to operate on larger sub blocks in order to take full advantage of the hardware. To back this statement with some numbers, we rerun the benchmark and set the sub block size to 4 KB. This should yield better results for both hardware and software AES, with an advantage for the former due to the already shown fact that blocks of this size are processed faster when using *omap-aes*.

Figure 5.11 shows the results and it is evident that our assumptions are correct. When using *ubi_crypt_convert* together with *omap-aes*, we observe rates between roughly 7.5 MB/s and 8.9 MB/s. In contrast, using *aes_generic* yields rates of only around 6.5 MB/s, which is still considerably faster than what we measured for a sub block size of 16 bytes. It looks however like the rate for *omap-aes* declines with increasing buffer size, which seems a bit counter-intuitive at first. To explain this, we need to recall that *ubi_crypt_convert* generates an

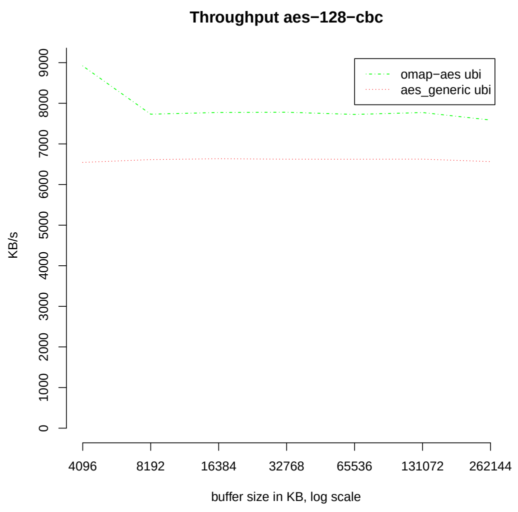


Figure 5.11.: Throughput, *ubi_crypt_convert*, 4K blocks

IV for every logical block it processes. Since we are working with a sub block size of 4 KB, there is only 1 single block for which we need to generate an IV for but this number increases proportionally to the buffer size. Each IV generation requires a single encryption of a 16 byte block, performed by *aes_generic* in every case. In addition, there is computational overhead from splitting the buffer into blocks, which increases with buffer size.

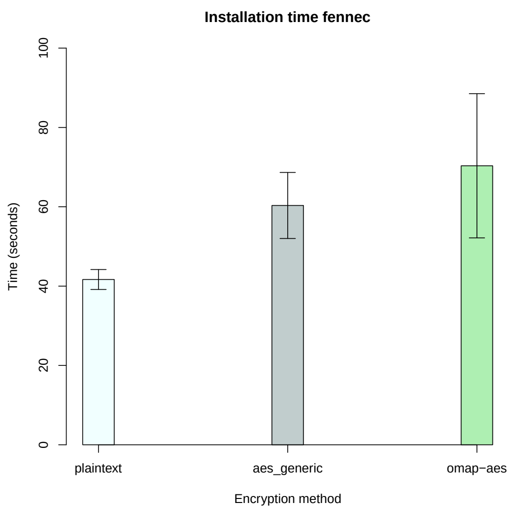
5.6. Performance: daily use

As indicated in Section 5.1.1, our test cases for daily use scenarios cover booting the OS and installing and running applications. We start by running all tests on an unencrypted device to get a set of reference values to which we can compare later. Afterwards, we consecutively encrypt our root partition and the other system partitions, rerunning the tests in between these steps. This section discusses the results.

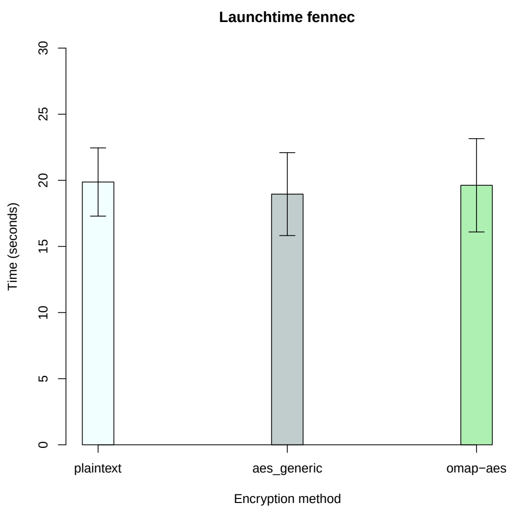
The first things we examine are the installation of a fairly large but common package, namely fennec[4], as well as the startup time of the installed application itself. Before we start the installation, we ensure that caches are flushed[5] to ensure that we start out with the same preconditions. Since download times can vary, we do not include this part of the installation and instead install the package from a local copy by directly invoking dpkg. The duration is recorded by means of the time command, which measures the execution time of a specific command. The startup time is measured in the same way, with the application being terminated as soon as its user interface becomes accessible. The time measured for application startup thereby coincides with the time required to start and terminate the application.

[4]This is Mozilla Firefox Mobile

[5]By rebooting the phone, issuing echo 1 > /proc/sys/vm/drop_caches to flush the caches proved unreliable.



(a) Mean of time required to install fennec



(b) Mean of time required to start fennec

Figure 5.12.: Measurements involving fennec

The installation process installs 259 files, which require a total of 40.03 MB. During startup, fennec and its child processes access 1057 files.

Figure 5.12a and 5.12b show two barplots depicting the mean of the time that was required to perform an operation involving fennec, Figure 5.12a representing the installation and Figure 5.12b representing the startup. Each barplot shows three different types of experiments that were performed, each with a different encryption method. We expected a high variance for these measurements due to the influence of other processes running on the system. The standard deviation differs significantly depending on the type of experiment. For our installation time benchmark, we observe a mean of 41.67s with an SD of 2.52s for a plain text root filesystem, 60.33s with an SD of 8.33s when using *aes_generic* for encryption of the root filesystem and 70.33s with an SD of 18.18s when using *omap-aes*. For our startup time benchmark, the mean is 19.51s with an SD of 2.58s for plaintext, 18.96s with an SD of 3.14s for *aes_generic* and 19.62s with an SD of 3.53s for *omap-aes*. Due to our small sample size of 5, it is impossible to draw accurate conclusions from this unfortunately. The general trend nevertheless seems to correspond to our previous filesystem benchmarks. It should not come as a surprise to see the shortest required time for using no encryption at all. Considering that our previous measurements showed *aes_generic* to be slightly faster than omap_aes due to small block sizes being employed, the same holds true for the observed time difference between omap_aes and *aes_generic* in this barplot.

Next, we take a look at the bootup times, both with an unencrypted and with an encrypted filesystem. As before, we distinguish between using *aes_generic* and *omap-aes* for encryption. We measure the time between pushing the power button and the first moment the desktop appears, using a stopwatch.

Figure 5.13 shows us a mean bootup time of 37s with an SD of 1s for an unencrypted root filesystem, 71.4s with an SD of 0.89s for an encrypted one using *aes_generic* and 66.6s with an SD of 0.89s for using *omap-aes*. The same restrictions for the accuracy as for the previ-

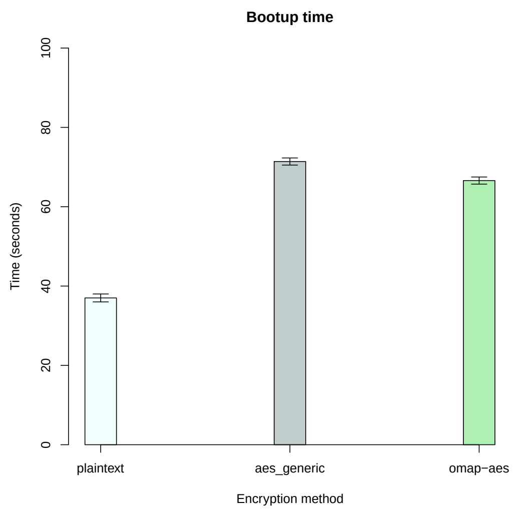


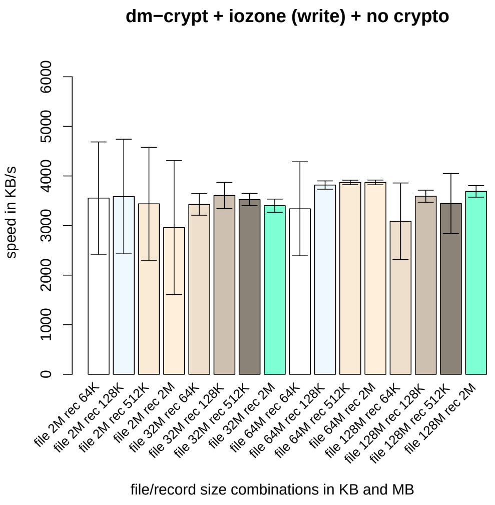
Figure 5.13.: Time required to boot

ous benchmark apply since we used the same sample size. It is nevertheless interesting to note that the standard deviation is much smaller and that *omap-aes* seems to fare better than *aes_generic*. The difference between between the two crypto modules is very small though and may very well be attributed to measurement inaccuracies.

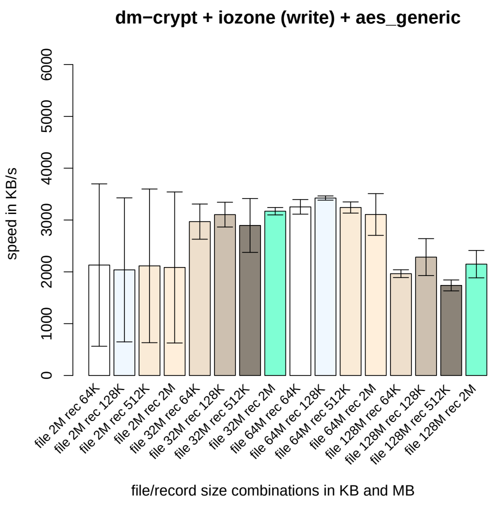
5.7. Performance: dm-crypt

We would like to take a very short look at dm-crypt now so that we can compare our implementation to an alternative approach. Of course, the two approaches are vastly different and have mainly one thing in common: they are both used for encrypting a device based on its logical blocks, on a layer between filesystem and device drivers[6]. The differences are described in more detail in Section 6.2. We use a partition on an external MMC to run our benchmarks, for which we choose ext3 as filesystem. Since we are interested in a direct comparison between encrypted UBI devices and dm-crypt volumes, we compare only iozone results with flushing times included and thereby ignore the filesystem cache as much as possible. To ensure that the results are as comparable to our UBI related results, all other benchmark parameters remain the same as used for UBI as well. Our first attempt uses no encryption at all and runs on a plain ext3 partition on the MMC card. Looking at Figure 5.14a, which shows the write rates for various file and block size combinations, we already see a significant difference to our UBI benchmarks. The MMC card is apparently much slower than the NAND device and we achieve an average write rate of 6.5 MB/s while the mean for an unencrypted UBI device lies at 26.7 MB/s. We then encrypt the MMC partition using dm-crypt and format it as ext3. Figures 5.14b and 5.14c show the same file and block size combinations again, this time for the encrypted device, once using *aes_generic* and once using *omap-aes*, respectively. We notice some characteristics which differ from UBI. Large files of

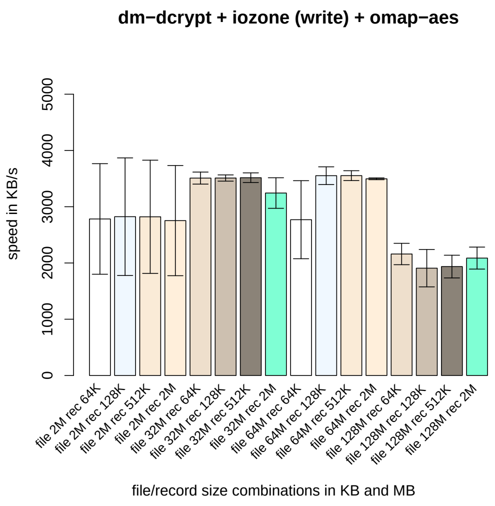
[6]This is of course simplified, there are additional layers in-between.



(a) dm-crypt, iozone, write, no encryption



(b) dm-crypt, iozone, write, *aes_generic*



(c) dm-crypt, iozone, write, *omap-aes*

Figure 5.14.: Dm-crypt with iozone, ext3

128 MB cause the write rates to drop by about 1 to 1.5 MB/s for dm-crypt with either encryption method. This effect cannot be seen once we enable flushing for UBI devices, however it is observed when relying on the filesystem cache. We have to keep in mind that we are dealing with a block device here which inherently uses an additional cache mechanism, the buffer cache. Its effect should be diminished by the use of flushing, however since we did not look into the Linux block I/O layer and the device-mapper sufficiently, we cannot make any definite statements here.
Overall, the write rates for encrypted dm-crypt volumes differ less from unencrypted volumes than it is the case for UBI devices. This is because the MMC card employed in our tests cannot write faster even without encryption. The device-mapper uses 512 byte blocks internally and we can recall from our throughput benchmarks in Section 5.2 that block sizes of 512 bytes yield encryption rates of only around 5.5 MB/s. This explains the observed maximum write rate 4 MB/s on dm-crypt volumes.
Obviously, reading benefits strongly from the filesystem cache which makes it harder to evaluate the actual dm-crypt performance without using files large enough to defeat the cache. Due to lack of time we do not discuss this in this thesis, the results can however be found in Appendix B.2.

5.8. Performance: conclusion

Now that we have looked in detail at the performance of our implementation and have drawn a comparison to an alternative solution, it's time to say a few concluding words. As we can see, results can differ significantly, depending on where and how one measures. For our performance measurements, we used three different layers to take samples from, deepest layer first: the cryptographic engine, the UBI function responsible for cryptographic conversions and the filesystem. Each higher layer depends on the lower layers. As we can see from the results, there are external factors playing an important role as well, namely page cache and block size. While it is possible to perform AES-128-CBC conversions at a rate of up to 9 MB/s on the hardware of the N900, the UBI layer can only take full advantage of this when using an optimal block size of 4 KB or larger. For the default block size on the N900 of 512 bytes, we end up with 3.5-5 MB/s. This translates into less than 3 MB/s for the filesystem layer since we have additional overhead from our writing strategies. Reading is not affected by this last issue and therefore is only limited by the block size used by UBI. For daily usage, the file system layer is the most important one and we have seen that the performance impacts of using an encrypted UBI device underneath UBIFS are reasonable, which is mainly owed to caching. Of course, the size of said impact depends highly on the specific use-case. For example, applications that do not want to rely on the integrated caching mechanisms and implement their own algorithms as well as applications that work with data sets which do not fit into the cache are obviously affected the most by it. While it seems unlikely that anyone would want to run a Database Management System (DBMS), which typically relies on direct I/O, on a smartphone and expect great performance, a more likely application would be a form of secure logging for which it is vital to know whether an entry was written to disk or not. Judging from our benchmark results, the file size boundary for noticeable slowdowns lies between 64 and 128 MB. On a standard N900 installation, there are very few system files that come even close to this boundary, one example being an 58 MB

file belonging to the icedtea6[7] package. To examine the typical file size distribution on the N900, we use a N900 which has been used on a daily basis for roughly a year. Aside from the standard system files, it contains user data than can typically be found on smartphones: photos and videos recorded with the integrated camera, audio files and a large number of custom-installed applications. We use the sizes of all files except for special files, such as device files or files belonging to the proc filesystem, to determine the file size probabilities. Of course, we rely on a sample size of one here, therefore the results do not bear any statistical significance. Let us take a look anyway. For the distribution of sizes for the 65939 files found, the 50% quantile lies at 2435 KB, the 75% quantile at 9247 KB and the 95% quantile at 61440 KB. Obviously, there are only very few large files on this particular installation. Note that this includes files from all partitions. Since the root partition is very small with 256 MB compared to the 27 GB *MyDocs* partition, it is not likely that large files such as multimedia files are stored on it. Therefore there is a good chance that we rarely reach the 64 MB boundary in daily use. Of course, there might be larger unmanaged flash chips built into mobile devices in the future.
As a bottom line, it is important to remember that we tested a prototype in this thesis for which there is still much room for improvements, especially performance-wise. Even configuration changes such as using a different cryptographic mode of operation might already cause noticeable changes. This may also be true for dm-crypt, which we only tested with the same cryptographic configuration as UBI. Since block devices are not the focus of this thesis, we do not go deeper into this but we take an outlook on possible future improvements to our prototype implementation in Section 7.3.

5.9. Power consumption

To measure power consumption, we use the values reported by hal-device, which is part of the HAL daemon package. This tool reports, amongst many other values, the current battery charge level in milliampere-hours (mAh). We use this information to measure the power consumption for 3 different scenarios:

- **idle** No user interaction, encrypted and unencrypted UBI rootfs
- **iozone active** Iozone running continuously on encrypted and unencrypted UBI partition
- **throughput benchmark** Continuous throughput benchmark

In contrast to the throughput benchmark, iozone also causes I/O access on the flash device. It can be expected that this requires more power than the pure block encryption, as done by the throughput benchmark. All tests involving cryptography are run with *omap-aes* and *aes_generic*, separately. Figure 5.17a and 5.17b show the respective results. For these figures, the x-axis shows the time in 5 second intervals and the y-axis shows the battery charge level in percent. As a reference for our power consumption measurements, Figure 5.15 shows the charge level on an idle N900, the x-axis representing the time in seconds. This means I/O access is reduced to a minimum and we can observe the discharging characteristics of the battery. Due to the long duration of this experiment, we took only one sample. We can

[7] A Java Development Kit (JDK) implementation, launched by Red Hat

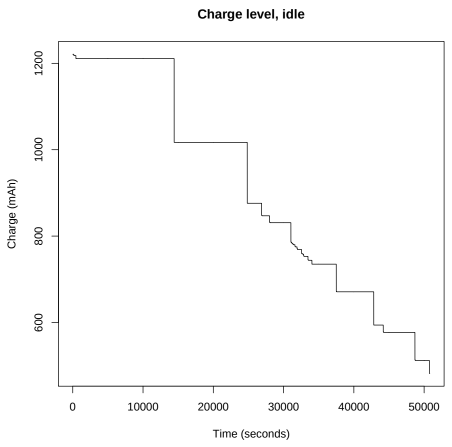


Figure 5.15.: Charge level while idle

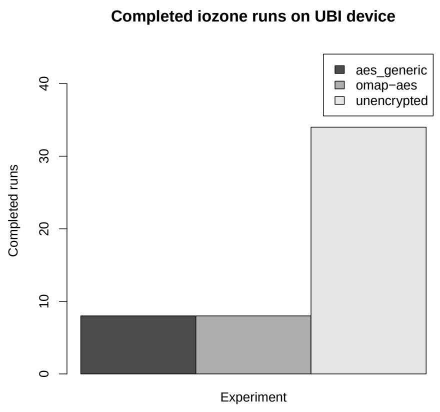
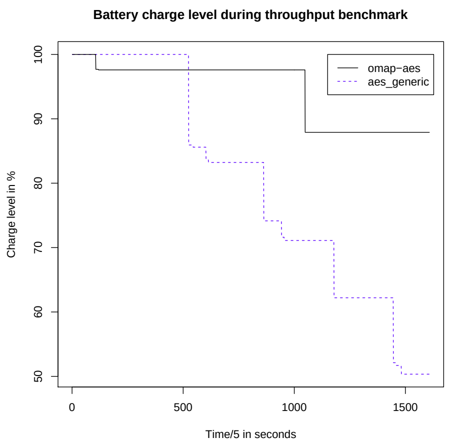
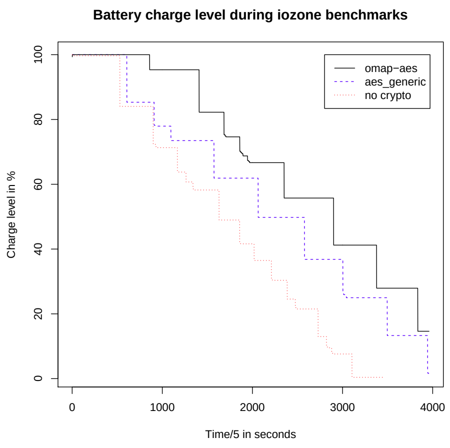


Figure 5.16.: Iozone runs on one battery charge



(a) Battery charge level, throughput benchmarks



(b) Battery charge level, iozone benchmarks

Figure 5.17.: Power consumption measurements

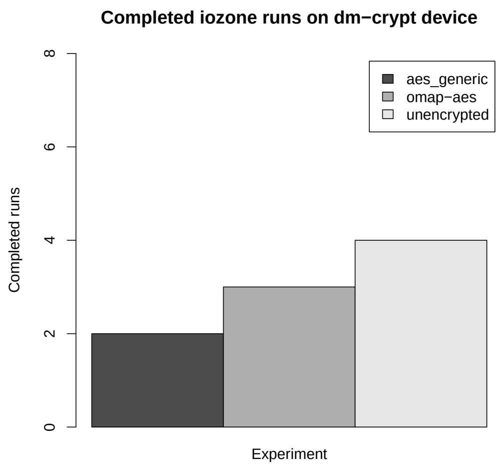


Figure 5.18.: Completed dm-crypt runs on one battery charge

observe an almost linear decline, however the graph is far from straight, which may be due to inaccuracies in the measurement or in the reports from HAL.
Figure 5.17a, which depicts the graphs for the battery charge level while running the throughput benchmark over a duration of 134 minutes, shows that using hardware acceleration for AES requires less power than employing the CPU. While the charge level does not drop below 87% when using *omap-aes*, it ends up at 50% with *aes_generic*. If we run iozone instead of the throughput benchmark, we also notice that charge levels drop faster for software based encryption. However, we also notice that using no encryption at all causes even faster depletion. Why is that? In the last paragraph, we explained that iozone causes I/O access on the flash device. We also observed in our previous benchmarks that using no encryption at all causes significantly faster I/O rates. Indeed, looking at Figure 5.16 we see that we complete 34 iozone benchmarks without encryption on one battery charge in contrast to 8 with encryption. As we mentioned before, the encryption and decryption process is slowed by the choice of block size and as long as the encryption rate is slower than the rate at which the data can be written to or read from the flash device, more power is used for I/O operations than for the cryptographic routines. As a result, we end up with an empty battery faster when not using encryption and at the same time manage to complete more iozone runs than when using encryption. It is also possible that flash I/O requires more power than encryption, especially when using the cryptographic co-processor. To confirm this, different measurements, which are not influenced by the UBI layer are required, however we do not perform these due to the lack of time. For comparison, Figure 5.18 shows the number of completed iozone runs on a block device, as described in Section 5.7. Just like the smaller gap we observed for write rates between unencrypted and encrypted device, there is a smaller difference for this number as well. We manage to complete only 4 iozone runs on an unencrypted device, which is much less than what we counted for unencrypted UBI devices. This suggests the MMC card requires more power per I/O access, however since we did not analyze dm-crypt in detail, we cannot make any definite statements here.

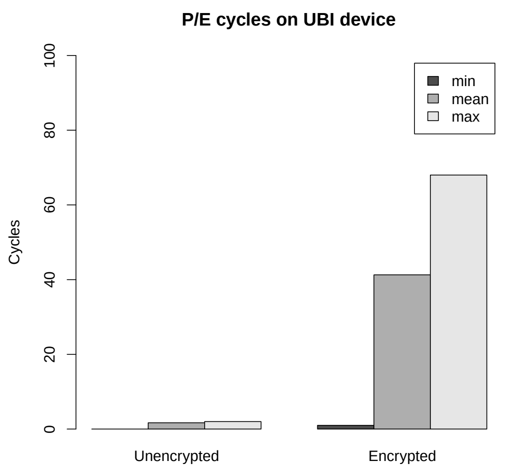


Figure 5.19.: P/E cycles for one iozone run

5.10. Flash endurance

For this experiment, we create a UBI device and UBIFS volume on a clean MTD device to ensure that all erase counters are zero before we start. We then run iozone on this volume, ensure that all data is flushed from the cache and then unmount the volume, taking note of the erase counters. We repeat the same process for an encrypted UBI device. For this experiment, we can be sure that repeating it with the same parameters will generally yield the same result because there is no external influence on our measurements. No other process writes to the UBI device and iozone always writes the same patterns when invoked with the same parameters. As we can see in Figure 5.19, the number of P/E cycles is significantly higher for encrypted UBI devices than for unencrypted ones. For the former, the maximum number of P/E cycles is 68, the mean is 41.28 and the minimum is 1. For the latter, the maximum is 2, the mean is 1.69 and the minimum is 0. That means that on average, there are 24.43 times as many P/E cycles for encrypted UBI devices. Even though this is a very large number, we should keep in mind here that iozone is designed to put a large amount of stress on a filesystem and that it is unlikely that anyone will run applications with high I/O load on the root partition of his phone. Nevertheless, this figure is important for the decision of whether the benefits of encrypting empty space outweigh the disadvantages. The decision depends on the individual case, which we cannot cover here.

6. Related work

In this section, we take a look at two related projects and draw comparisons to our work.

6.1. WhisperYAFFS

In parallel to our work, WhisperSystems developed and released experimental patches for adding filesystem-level encryption to YAFFS, called WhisperYAFFS [35]. YAFFS is the primary filesystem used by Android devices for unmanaged flash devices. WhisperYAFFS is integrated into the WhisperCore, a "secure Android platform dedicated to providing the security and management features necessary for transforming a consumer phone into an enterprise-class device" [36]. In its core, WhisperYAFFS works very similar to our UBI encryption layer. Unlike UBIFS, which does not directly operate on MTD devices but relies on UBI instead, YAFFS talks directly to the MTD layer. It uses page-sized blocks as a management unit instead of PEBs, which are written consecutively to a PEB until it is full. Then the next PEB is selected, until the number of free PEBs drops beneath a certain threshold, which triggers the garbage collection. Whenever a page is written to or read from the underlying device, WhisperYAFFS converts it using AES-XTS. The position of each block in NAND memory is used as so-called tweak[1]. The continuous writing of flash in fixed block sizes is better suited for block ciphers than buffers of arbitrary size that are processed by UBI's read and write functions. Unlike our solution, no other filesystems are supported due to the tight integration into YAFFS.

6.2. Dm-crypt

In this work, we have compared UBI to dm-crypt, which is a device-mapper target in the Linux kernel. It provides support for transparent encryption of block devices using the crypto API [37]. It is in so far closely related that it does the same work for a different type of device and indeed, much of our code is inspired by it. Nevertheless, there are some notable differences in the two systems. Most significantly, device-mapper I/O requests are always composed of blocks of 512 bytes. The encryption process can be deferred, which allows dm-crypt to organize its work into queues and split it up between the available CPUs. This can result in better performance on multi-core systems. Since the first smartphones using these are already available, this is an interesting aspect. Our modifications to *ubiattach* were inspired by cryptsetup, which is not part of dm-crypt but was written to complement it.

[1]The tweak is essentially equivalent to the IV used by AES-CBC

7. Future work

A few open questions and tasks remain, which should be looked into. As we have shown, our design and implementation are far from perfect. There is a lot of potential for optimizations and solutions that address the existent problems and improve performance. Furthermore, there are a few points that could not be addressed because they either lay out of scope of the thesis or collided with time constraints.

7.1. Open questions and problems

During our flash endurance measurements we realized that we made a mistake in our implementation. We have to erase PEBs before rewriting them and we do so by calling *do_sync_erase* in *ubi_io_write_crypted_aligned*. This erase function does not update the erase counters however. Since we are interested in accurate erase counters and because this also results in inconsistent information for the wear leveling, we decided to replace the *do_sync_erase* with *sync_erase* from the wear leveling system. This function ensures consistency with some but unfortunately not all relevant data structures dealing with erase counters. This is because these structures are normally updated before *sync_erase* is called. We ran out of time to fix our implementation in this regard and decided to track erase counters separately instead. We do so by incrementing our own counters during the invocation of *do_sync_erase*, which is the only and final function that calls the actual MTD erase routine. Therefore we can be sure that every block erase is accounted for. While this gives us the accurate picture of the number of additional P/E cycles on encrypted devices which we wanted, it is possible that providing inaccurate EC values to the wear leveling system influences the performance characteristics we looked at so far. It is important that this fix is completed in the future and that our results are reevaluated under the new conditions.

7.2. Measurement methods improvements

The statistical significance of our measurements suffers from a lack of samples. The long duration of some of our benchmarks and especially the recharge cycles for power measurements made us rely on estimates based on very small sample sizes at times. This affected above all the accuracy of our filesystem, typical use and power consumption benchmarks, which displayed a high level of variance. This is no surprise, considering that they were performed on a multi-tasking capable system, which had other processes competing for the same resources. While identifying and eliminating these processes is an option for refining our results, it would be a tailoring of our results towards an optimum that is unlikely to occur under normal use conditions. It is however our intention to model our experiments as

closely to reality as possible, which means our only option to increase confidence is a larger sample size.
There are also further aspects which have not been covered in this work. Even though we tried to stick as closely as possible to realistic scenarios, it would be worthwhile to go further in this direction and investigate the effects on actual daily usage. Last but not least, repeating our tests with different cryptographic algorithms and modes of operation is another interesting point left open.

7.3. Performance and flash wear improvements

As we have shown in Section 5, our design and our proof-of-concept implementation do not perform as well as they could, provided a few changes were made. Some of these changes are comparatively easy to implement, others are more complicated or require certain trade-offs. These improvements are mainly related to performance and flash lifetime. During the evaluation of our measurements, we noticed that block size plays an important role with respect to how fast data can be read from and written to encrypted UBI devices. Cryptographic conversion is more efficient for larger block sizes but while flash devices are often organized in large PEBs, these blocks are frequently written in small increments. Due to our design choices, we are forced to write full PEBs, which negatively affects both flash lifetime and performance. By caching small writes and combining them into larger writes wherever possible, we could alleviate these two problems. This would either require a separate cache layer or extensive modifications to UBI. Determining which of these two options is the better choice and how much we benefit from it is an interesting topic for future work.
Another potential improvement is the change from CBC to XTS. This would reduce the overhead we experience from chaining by not forcing us to re-encrypt subsequent blocks whenever we modify a sub block. Newer devices than the N900 might be built with a SoC supporting this mode of operation in the future.
Last but not least, our measurements were done using a prototype implementation, not a finished product. There is still much room for smaller corrections and improvements in the code, like for example caching block IVs, which need to be calculated for every I/O access.

7.4. Hardware accelerated hashing

The OMAP 3430 chipset installed in the N900 also provides hardware support for the SHA-1 and MD5 hashing algorithms. Due to time constraints, we did not examine the impact of using the SHA-1 acceleration in this thesis. It is unlikely however that this would have affected our results because hashing is only used for the initial setup of the encryption context for the ESSIV generation in AES-CBC-ESSIV. Nevertheless, it would be interesting to determine whether the hashing unit of the OMAP 3430 shows similar characteristics as the encryption unit, power consumption and performance-wise. There are kernel layers other than the filesystem layer which make use of hashing algorithms as well and the interface could also be exported to user space to allow applications like openssl to take advantage of it. Sadly, only SHA-1 is supported which is known to be weak against certain types of collision attacks [38]. It is likely that support for additional, more secure algorithms is added in future devices however.

7.5. Power efficiency

We suggested the possibility of flash I/O requiring more power than the cryptographic coprocessor in Section 5.9, which we did not confirm. This is particularly interesting for optimizations regarding power efficiency of our design. Our design decisions result in larger I/O loads and it would be interesting to review them under the aspect of saving power. Since we discovered this property late during the work of our thesis, it is not part of it, however it is an important point for future work.

7.6. Secure key storage

Cryptography always comes with the requirement of secure key storage. One common approach is to have the user enter a secret that decrypts the locally stored key from its storage, which is also the approach discussed previously in this thesis. Many devices, not limited to smartphones, provide a secure memory location nowadays that can be used for storing key material without directly exposing it to the user or the operating system, like smart cards. Many ARM CPUs, including the one installed in the N900, also provide a security extension called TrustZone, which provides two modes, a secure and a ”normal” one, between which the processor can switch. A minimal secure kernel can run in secure mode, which provides access to data that is inaccessible to the standard kernel running in ”normal” mode [39]. Together with secure boot [40], TrustZone can be used to ensure that not even the user has access to the key storage, which is why the most common application for this is Digital Rights Management (DRM). Without doubt, running the device encryption within the secure kernel would provide the highest amount of security and significantly lower the risk of the encryption key being lost or leaked. Of course, it would require the user to put sufficient amounts of trust into the device manufacturer who would inevitably be the only entity having access to the crypto keys.

8. Conclusion

In the course of this thesis, we have designed, implemented and evaluated a system that allows the encryption of unmanaged flash devices on systems running a Linux kernel. We have presented solutions for many of the problems associated with this task and evaluated our implementation under various aspects. Our findings suggest that the encryption of unmanaged flash devices is conceptionally similar to block device encryption, albeit with a few notable differences, due to the way flash memory is written. Some of them required us to find alternative solutions, which incurred the need for trade-offs either in terms of security or in terms of performance. We decided to sacrifice performance for security and to determine the impact on performance and usability that came with this decision. Since our target platform is mainly represented by mobile devices, we also had to examine related aspects, such as changes in power consumption and key entry.
We expected a large performance loss, owed to our design decisions towards security and found this expectation partially met by our benchmark results. As we have shown however, the impact of this loss depends on the use case scenario and a number of factors. These factors can be split into the three categories filesystem related, cryptography related and hardware related. We have determined the main filesystem related factors to be caching, block[1] size and file size. Regarding cryptography, we suspect that using a different algorithm like AES-XTS could bring performance improvements in some situations, however we have not tested this due to lack of time. The other important advantage of AES-XTS over AES-CBC-ESSIV is the lack of weaknesses against some attacks, such as content leaks, malleability and modification detection, as we explained in Section 2.1.3. Should upcoming chipsets support this algorithm in hardware, it would be very interesting to see a comparison under the same criteria we used in this thesis.
The underlying hardware plays an important role in such that it dictates the way data can be stored and accessed and related parameters, such as the PEB size. As we have seen, larger PEB sizes mean more overhead when writing in our design. We can also conclude that employing the cryptographic co-processor in our benchmarks results in reduced power consumption and, for certain conditions, better performance compared to software based encryption. In particular, we observed a performance dependency on the block size, for which we determined the optimal values. This knowledge is valuable for future research and can be used to alter and improve our design and implementation.

For our particular use-case of an encrypted UBI device as the root partition on a Nokia N900 smartphone, we have seen that the effect on user experience is not necessarily as pronounced as measurement numbers may suggest at first. User experience is however a quality hard to measure, therefore this is only a very subjective assessment. If we look exclusively at the numbers, we see that the root device encryption causes slight delays during bootup,

[1]Note that a block in this context refers to the unit of access to files stored on the filesystem.

application startup and drains the battery faster under load. In the particular case of the Nokia N900, the root partition is fairly small and the system and most of its applications are configured to store data on a different partition. This means that the majority of file accesses on the root partition is reading, for which the performance penalty of encryption is lower than for writing, as we have seen. Of course, even more volatile data could be moved to separate partitions to further limit the number of writes to the encrypted UBI devices, however we should keep in mind that this data is likely to be the very confidential data that we meant to secure in the first place. Fortunately, for this scenario dm-crypt offers itself as a well-known solution. We used dm-crypt in combination with an MMC partition for a short comparison between managed flash and unmanaged flash encryption performance under similar conditions. While we measured a significantly smaller slowdown from encryption while writing, this was mainly due to the comparatively slow write speed of the MMC card used in our experiments. In fact, the write rates observed for an unencrypted partition were fairly close to our measured average encryption speed on the N900 for the block size used by dm-crypt. This suggests that the maximum speed for encrypted block devices has already been achieved, unless the block size is changed and consequently that the gap between write rates will grow further for faster MMCs. Since our benchmarks tried to ignore caching effects on the results as much as possible, our statements about the expected influence on user experience in scenarios involving caching are likely to hold true for dm-crypt as well. This point was not examined in detail, as it goes well beyond the scope of this thesis.

Using device encryption means additional work for either the CPU or the cryptographic co-processor, which obviously requires more power. We determined that using the dedicated hardware for cryptographic operations is more efficient in terms of power consumption than using the CPU. Unsurprisingly, we managed to complete less work on one battery charge when operating on an encrypted UBI device than on an unencrypted one. It remains a point for the future to find ways for making our work more power efficient.

We also looked at flash endurance, expecting considerably more P/E cycles on encrypted UBI devices due to our design choices. Unfortunately, this expectation was met as well and we ended up with more than 24 times as many P/E cycles. It remains an open question how far the reduced lifetime of the flash chip affects the user, since the I/O load depends entirely on the individual use-case. Taking smartphones as an example, which are replaced by new versions in relatively short time frames, it is not unlikely that the device is replaced before the flash chip becomes unusable. Nevertheless, it is desirable to reduce the flash wear and increase the lifetime of the device. We discussed possible options for this along with performance optimizations, as the two are partially related. These changes go beyond the prototype implementation we discussed in this work and remain a topic for future work.

Even though our work is far from being a perfect solution, we believe that it provides a valuable insight on the problems related to the encryption of unmanaged flash devices and outlines different approaches to solving them. It provides a detailed overview of the advantages and disadvantages of our chosen design and analyzes how well it performs when put into practice. On its own, this thesis can be used as a solid base for securing non-volatile memory on smartphones and possibly other mobile devices.

Bibliography

[1] http://www.gartner.com/it/page.jsp?id=1421013, retrieved April 2011

[2] Smartphones Changing the Way Business Professionals Work and Live, http://blog.ringcentral.com/2010/04/smartphones-changing-the-way-business-professionals-work-and-live.html, April 2010

[3] Stored Data Security, http://us.blackberry.com/ataglance/security/features.jsp#tab_stored_data, retrieved April 2011

[4] iOS 4: Understanding data protection, http://support.apple.com/kb/HT4175, retrieved April 2011

[5] Whisper Systems http://www.whispersys.com/, retrieved August 2011

[6] Limitations of Data Protection in iOS 4, http://anthonyvance.com/blog/forensics/ios4_data_protection/, retrieved April 2011

[7] BlackBerry encryption 'too secure': National security vs. consumer privacy, http://www.zdnet.com/blog/igeneration/blackberry-encryption-too-secure-national-security-vs-consumer-privacy/5732, July 2010

[8] India Is Unsatisfied with RIMs Encryption Solution, http://www.blackberrymanual.net/india-is-unsatisfied-with-rims-encryption-solution.html, March 2011

[9] SAMSUNG Semiconductor - Products - Fusion Memory - OneNAND http://www.samsung.com/global/business/semiconductor/products/fusionmemory/Products_FAQs_OneNANDGeneral.html, retrieved May 2011

[10] MLC vs. SLC NAND Flash in Embedded Systems, http://www.smxrtos.com/articles/mlcslc.pdf, Yingbo Hu and David Moore

[11] Random versus Ecrypted Data, http://opensource.dyc.edu/random-vs-encrypted, retrieved April 2011

[12] Quality of Encryption Measurement of Bitmap Images with RC6, MRC6, and Rijndael Block Cipher Algorithms, Nawal El-Fishawy, Osama M. Abu Zaid and Nawal El-Fishawy, Nov 2007

[13] New Methods in Hard Disk Encryption, Clemens Fruhwirth, 2005

[14] Linux hard disk encryption settings, Clemens Fruhwirth, http://clemens.endorphin.org/LinuxHDEncSettings, retrieved June 2011

[15] P1619 Meeting, http://ieee-p1619.wetpaint.com/page/P1619%20Aug%2030th%2C%202006, Aug 30th, 2006

[16] IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices, http://ieeexplore.ieee.org/servlet/opac?punumber=4493431

[17] Understanding the Linux Kernel, Understanding the Virtual File System, Daniel P. Bovet and Marco Cesati Ph.D.

[18] TKS1 - An anti-forensic, two level, and iterated key setup scheme http://clemens.endorphin.org/publications, Clemens Fruhwirth, 2004

[19] bonnie++ http://freshmeat.net/projects/bonnie/, retrieved May 2011

[20] oprofile about page, http://oprofile.sourceforge.net/about/, retrieved May 2011

[21] Hardware Performance Counter Basics http://perfsuite.ncsa.illinois.edu/publications/LJ135/x27.html, retrieved May 2011

[22] Device-mapper Resource Page, http://sourceware.org/dm/, retrieved April 2011

[23] The advance of the ARM Cortex A8 and Cortex A5 http://ancillotti.hubpages.com/hub/The-advance-of-the-ARM-Cortex-A8-and-Cortex-A5, retrieved August 2011

[24] Nokia N900 Commented Hardware specs http://natisbad.org/N900/n900-commented-hardware-specs.html, retrieved June 2011

[25] Block device emulation over UBI, http://lists.infradead.org/pipermail/linux-mtd/2008-January/020381.html, January 2008

[26] An Introduction to NAND Flash and How to Design It In to Your Next Product, http://www.micron.com/document_download/?documentId=145, Micron Technology, Inc.

[27] Samsung fabs Apple A5 processor http://www.eetimes.com/electronics-news/4213981/Samsung-fabs-Apple-A5-processor, retrieved May 2011

[28] Bonnie++ http://www.coker.com.au/bonnie++/

[29] UBI - Unsorted Block Images http://www.linux-mtd.infradead.org/doc/ubi.html#L_ubidoc, retrieved May 2011

[30] Linux Device Drivers, Third Edition, Memory Mapping and DMA, http://lwn.net/images/pdf/LDD3/ch15.pdf, Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman

[31] Acceleration of encrypted communication using co-processors, http://diploma-thesis.siewior.net/html/diplomarbeit.html, Sebastian Siewior, 2007

[32] Driver porting: completion events, http://lwn.net/Articles/23993/, retrieved May 2011

[33] AES-128 CIPHER. MINIMUM AREA, LOW COST FPGA IMPLEMENTATION, M. C. LIBERATORI and J. C. BONADERO, 2007

[34] Test Details http://www.coker.com.au/bonnie++/readme.html, retrieved July 2011

[35] WhisperYAFFS, https://github.com/WhisperSystems/WhisperYAFFS/wiki, retrieved August 2011

[36] WhisperCore https://github.com/WhisperSystems/WhisperYAFFS/wiki, retrieved August 2011

[37] dm-crypt: a device-mapper crypto target `http://www.saout.de/misc/dm-crypt/`, retrieved June 2011

[38] Classification and Generation of Disturbance Vectors for Collision Attacks against SHA-1, Stphane Manuel, April 2011

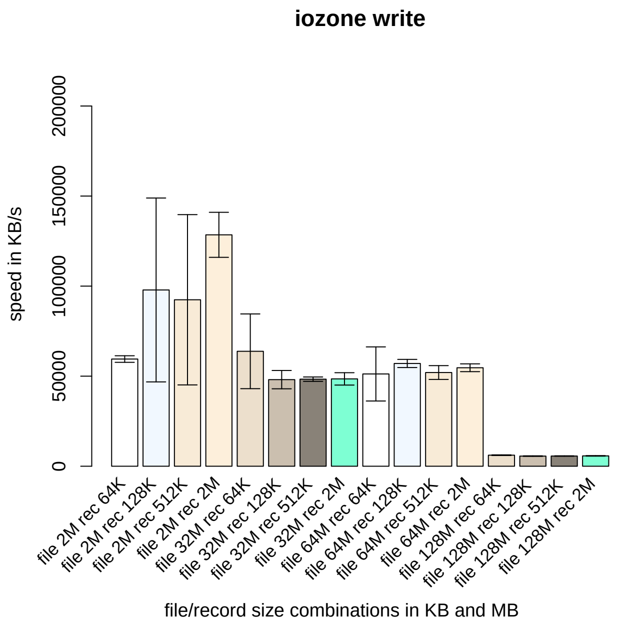
[39] What is TrustZone? `http://infocenter.arm.com/help/topic/com.arm.doc.faqs/ka6748.html`, retrieved September 2011

[40] ARM Security Technology Building a Secure System using TrustZone Technology, `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/CACGCHFE.html`, retrieved September 2011

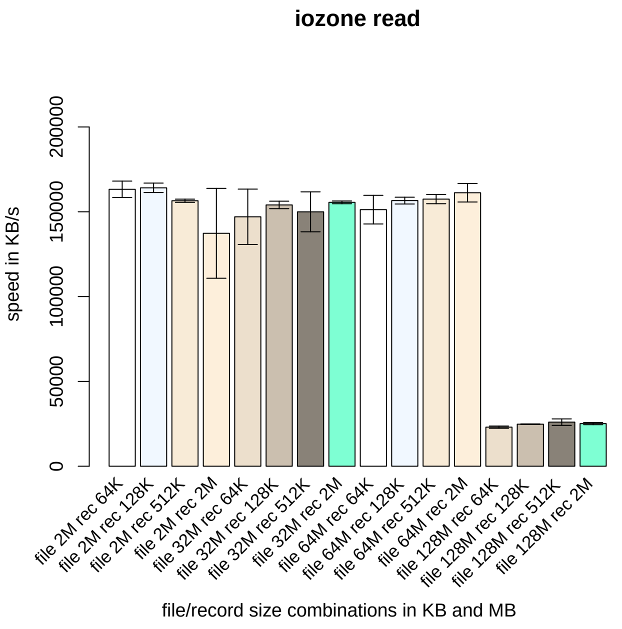
A. Nokia N900

B. Additional figures

B.1. UBI - iozone



(a) write



(b) read

Figure B.1.: omap-aes, no flushing

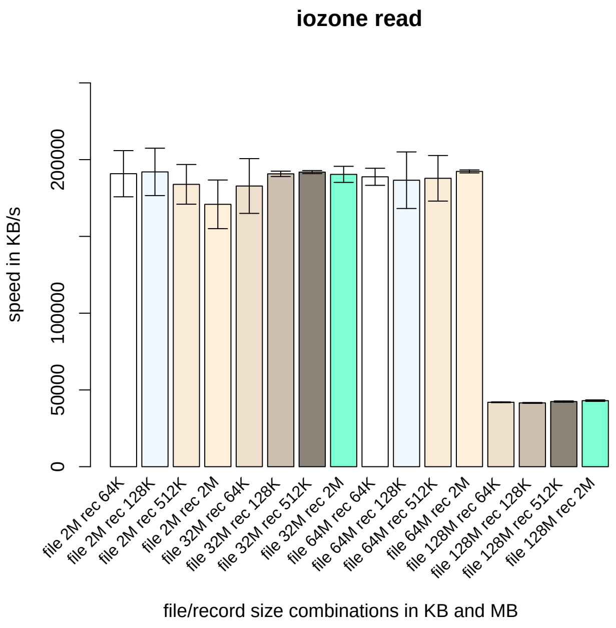
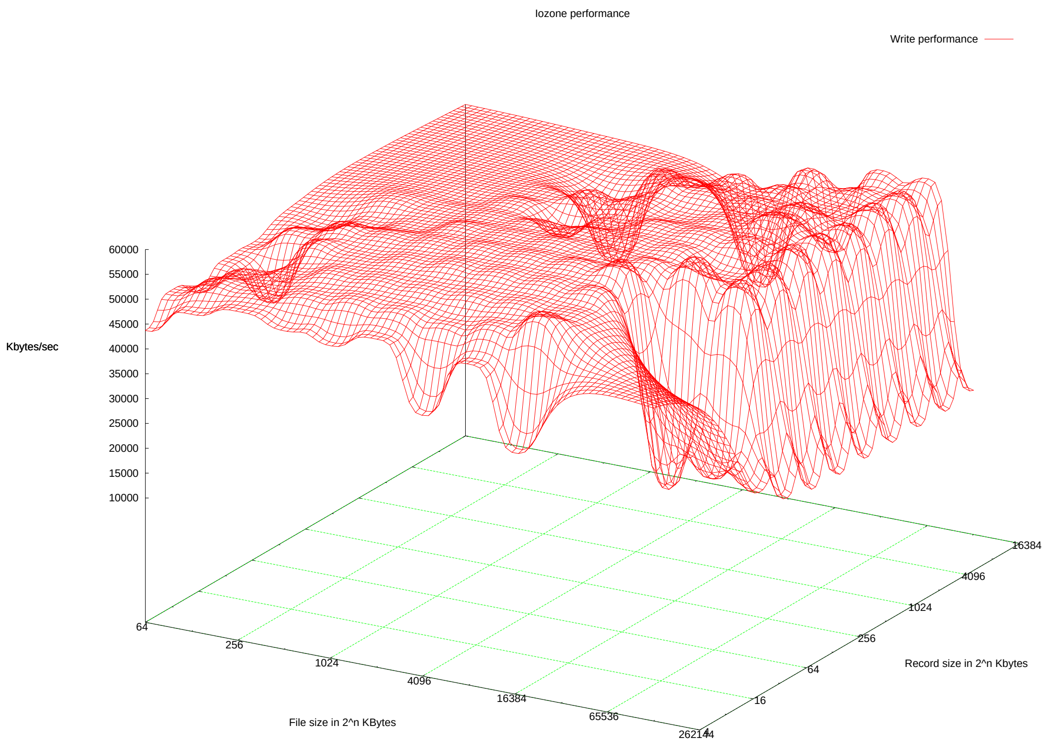
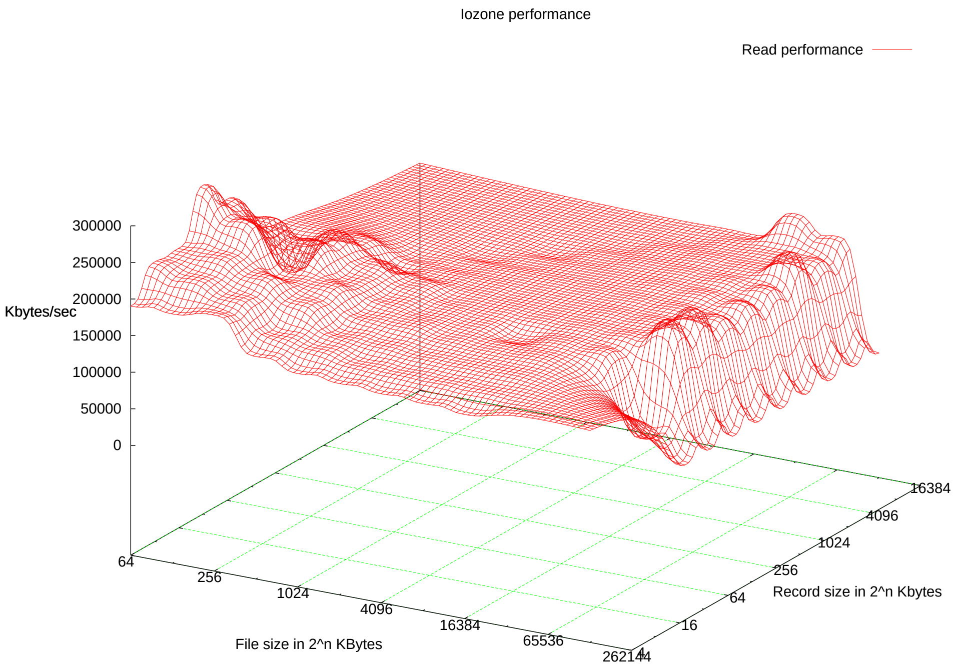


Figure B.2.: plaintext

B.2. Dm-crypt - iozone

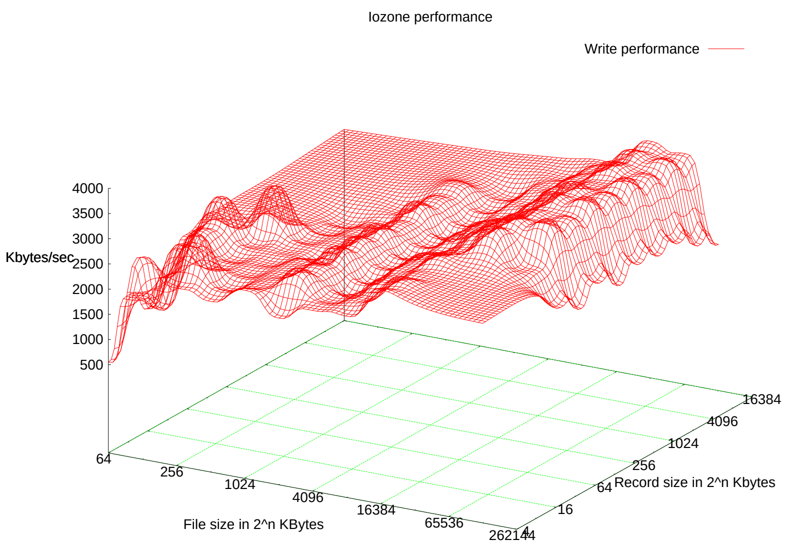


(a) write

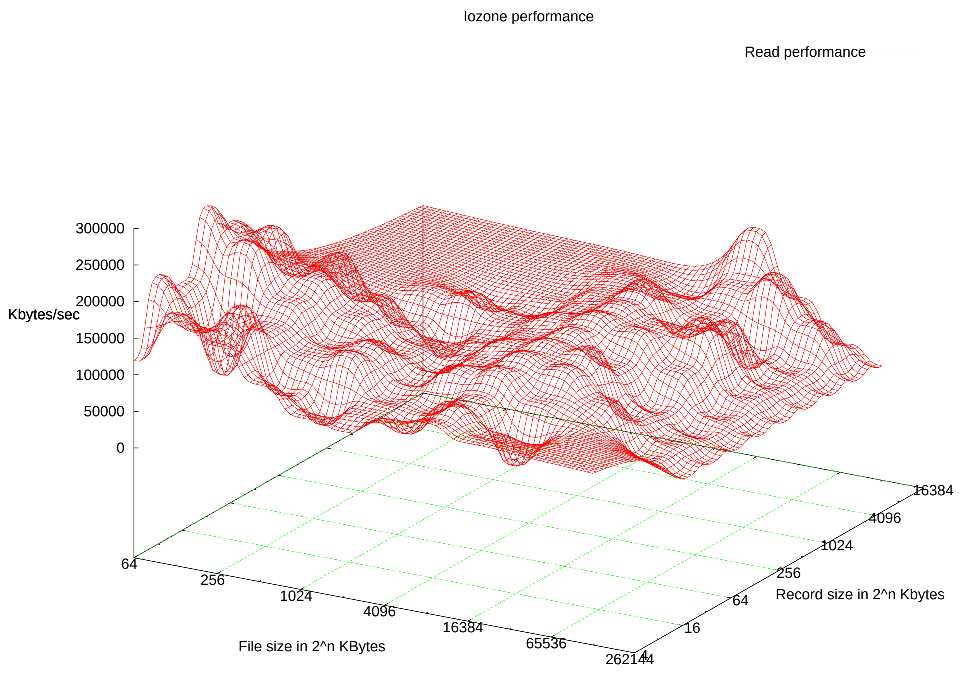


(b) read

Figure B.3.: dm-crypt with iozone, ext3, plaintext, flush



(a) write



(b) read

Figure B.4.: dm-crypt with iozone, ext3, *omap-aes*, flush

C. Source code

Listing C.1: ubi_crypt_convert

```c
int ubi_crypt_convert(const struct ubi_device *ubi, void *buf,
                              void *dst, size_t count,
                              int pnum, int offset, int dir)
{
        struct ubi_crypt_result result;
        int i, len, block_size = ubi->hdrs_min_io_size;
        int block_num, err = 0;
        int nents, tmp_len, rest;
        size_t count_save = count;
        struct scatterlist *sgentry;
        struct sg_table sg_tbl;
        struct ablkcipher_request *req;
        struct ubi_crypt_config *ucc = ubi->crypt_cfg;
        gfp_t gfp_mask = GFP_NOIO;
        u8 iv[ucc->iv_size];

        ubi_assert(is_power_of_2(block_size));

        if (count < block_size) {
                printk("ubi_crypt_convert:␣buffer␣must␣be␣larger␣or␣equal␣to␣hdrs_min_io_size,␣"
                        "is␣%zu␣but␣should␣be␣at␣least␣%d\n", count, block_size);
                return -EINVAL;
        }

        len = count > block_size ? block_size : count;

        block_num = ALIGN(offset, block_size) >> ucc->block_shift;
        if (ALIGN(offset, block_size) != offset) {
                printk("ubi_crypt_convert:␣offset␣must␣be␣aligned␣to␣block_size");
                return -EINVAL;
        }

        /* use pnum + block_num for IV */

        init_completion(&result.restart);

        req = mempool_alloc(ucc->req_pool, GFP_NOIO);

        ablkcipher_request_set_tfm(req, ucc->tfm);
        ablkcipher_request_set_callback(req, CRYPTO_TFM_REQ_MAY_BACKLOG |
                                             CRYPTO_TFM_REQ_MAY_SLEEP,
                                             ubi_crypt_async_done, &result);

        while (count) {
                //cond_resched();

                err = crypt_iv_essiv_gen(ucc, iv, pnum, block_num);
                if (err < 0) {
                        printk("Error␣generating␣IV\n");
                        goto alloc_err;
                }

                nents = len >> PAGE_SHIFT;
                rest = len % PAGE_SIZE;
                if (rest)
                        nents++;

                err = sg_alloc_table(&sg_tbl, nents, gfp_mask);
                if (err < 0) {
                        printk("Error␣calling␣sg_alloc_table\n");
                        goto alloc_err;
                }

                for_each_sg(sg_tbl.sgl, sgentry, nents, i) {
                        tmp_len = (i == nents - 1 && rest) ?
                                                rest : PAGE_SIZE;

                        sg_set_page(sgentry,
                                        mempool_alloc(ucc->page_pool, gfp_mask),
                                        tmp_len,
                                        0);
                }

```

```
		sg_copy_from_buffer(sg_tbl.sgl, nents, buf, len);

		err = do_convert(sg_tbl.sgl, sg_tbl.sgl, &iv[0], len,
				 req, dir, &result, ucc->iv_size);

		sg_copy_to_buffer(sg_tbl.sgl, nents, dst, len);

		for_each_sg(sg_tbl.sgl, sgentry, nents, i) {
			mempool_free(sg_page(sgentry), ucc->page_pool);
		}

		sg_free_table(&sg_tbl);

		if (err) {
			printk("do_convert_error:_%d\n", err);
			goto alloc_err;
		}

		count -= len;
		buf += len;
		dst += len;
		block_num++;
		len = count > block_size ? block_size : count;
	}
alloc_err:
	mempool_free(req, ucc->req_pool);
	return err ? err : count_save - count;
}
EXPORT_SYMBOL_GPL(ubi_crypt_convert);
```

Listing C.2: do_convert

```
static int do_convert(struct scatterlist *sg_in, struct scatterlist *sg_out, char *iv,
		      int blklen, struct ablkcipher_request *req, int dir,
		      struct ubi_crypt_result *result, int iv_size)
{
	int err = 0;

	ablkcipher_request_set_crypt(req, sg_in, sg_out, blklen, iv);

	if (dir == WRITE)
		err = crypto_ablkcipher_encrypt(req);
	else
		err = crypto_ablkcipher_decrypt(req);

	switch (err) {
	case 0:
		// success
		break;
	case -EINPROGRESS:
		// fall through
	case -EBUSY:
		wait_for_completion(&result->restart);
		INIT_COMPLETION(result->restart);
		err = 0;
		break;
	default:
		// encryption failed
		break;
	}

	return err;
}
```

Listing C.3: ubi_io_write_crypted_aligned

```
int ubi_io_write_crypted_aligned(struct ubi_device *ubi, void *buf,
				 int pnum, int offset, int len, size_t *written,
				 int canoverwrite, int erase)
{
	int converted, err;
	int orig_len = len;

	int aloffset = ALIGN(offset, ubi->hdrs_min_io_size);
	int alen = len;
	void *tmpbuf = 0;
	loff_t addr;

	if (aloffset != offset)
		aloffset -= ubi->hdrs_min_io_size;

	alen = ALIGN((offset - aloffset) + len, ubi->hdrs_min_io_size);

	if (NULL == ubi->crypt_cfg)
		return -EINVAL;

```

```
        *written = 0;

        tmpbuf = mempool_alloc(ubi->crypt_cfg->peb_pool, GFP_NOFS);
        if (ubi->crypt_cfg->peb_pool->curr_nr < 4)
                printk("Only %d elements in pool!\n", ubi->crypt_cfg->peb_pool->curr_nr);
        if (!tmpbuf) {
                err = -ENOMEM;
                ubi_err("Failed to allocate %d bytes when writing"
                        " PEB %d:%d\n", ubi->peb_size, pnum, offset);
                goto out_free;
        }

        /* read encrypted block without decrypting it */
        err = ubi_io_crypt_read(ubi, tmpbuf, pnum, 0,
                                ubi->peb_size, 0, 0);

        if (err && err != UBI_IO_BITFLIPS)
                goto out_free;

        /* updating partial block. decrypt block and update it */
        if (aloffset != offset || alen != len) {
                converted = ubi_crypt_convert(ubi, tmpbuf + aloffset, tmpbuf + aloffset,
                                        alen, pnum, aloffset, READ);

                if (converted < alen) {
                        ubi_err("ubi_crypt_convert failed with %d", converted);
                        err = -EIO;
                        goto out_free;
                }

        }

        memcpy(tmpbuf + offset, buf, len);

        converted = ubi_crypt_convert(ubi, tmpbuf + aloffset,
                                tmpbuf + aloffset, alen, pnum,
                                aloffset, WRITE);
        if (converted < alen) {
                ubi_err("ubi_crypt_convert failed with %d", converted);
                err = -EIO;
                goto out_free;
        }

        if (erase) {
                err = do_sync_erase(ubi, pnum, 0);
                if (err)
                        goto out_free;
        }

        /* we always write a full block */
        offset = 0;
        addr = (loff_t)pnum * ubi->peb_size;
        len = ubi->peb_size;

        err = ubi->mtd->write(ubi->mtd, addr, len, written, tmpbuf);

        if (*written == len)
                *written = orig_len;
out_free:
        if (tmpbuf)
                mempool_free(tmpbuf, ubi->crypt_cfg->peb_pool);

        return err;
}
```

Listing C.4: ubi_io_crypt_read

```
#if defined(CONFIG_MTD_UBI_CRYPTO) || defined(CONFIG_MTD_UBI_CRYPTO_MODULE)
int ubi_io_read(const struct ubi_device *ubi, void *buf, int pnum, int offset,
                int len)
{
        return ubi_io_crypt_read(ubi, buf, pnum, offset, len, 1, 0);
}

int ubi_io_crypt_read(struct ubi_device *ubi, void *buf, int pnum, int offset,
                int len, int crypt, int recursive)
#else
int ubi_io_read(const struct ubi_device *ubi, void *buf, int pnum, int offset,
                int len)
#endif
{
        int err, retries = 0;
        size_t read;
        loff_t addr;
#if defined(CONFIG_MTD_UBI_CRYPTO) || defined(CONFIG_MTD_UBI_CRYPTO_MODULE)
        void *tmpbuf = 0;
#endif

```

```
	dbg_io("read␣%d␣bytes␣from␣PEB␣%d:%d", len, pnum, offset);

	ubi_assert(pnum >= 0 && pnum < ubi->peb_count);
	ubi_assert(offset >= 0 && offset + len <= ubi->peb_size);
	ubi_assert(len > 0);

	err = paranoid_check_not_bad(ubi, pnum);
	if (err)
		return err > 0 ? -EINVAL : err;

	addr = (loff_t)pnum * ubi->peb_size + offset;
#if defined(CONFIG_MTD_UBI_CRYPTO) || defined(CONFIG_MTD_UBI_CRYPTO_MODULE)
	if (NULL != ubi->crypt_cfg && crypt) {
		int alen = ALIGN(len, ubi->hdrs_min_io_size);
		int aloffset = ALIGN(offset, ubi->hdrs_min_io_size);
		int offdiff = 0, olen = len;

		if (alen != len || aloffset != offset) {
			/* unaligned access */
			if (aloffset != offset) {
				aloffset -= ubi->hdrs_min_io_size;
				offdiff = offset - aloffset;
				len = len + offdiff;
				/* length changed */
				alen = ALIGN(len, ubi->hdrs_min_io_size);
			}

			len = (alen != len) ? alen : len;

			tmpbuf = mempool_alloc(ubi->crypt_cfg->peb_pool, GFP_NOFS);
			if (ubi->crypt_cfg->peb_pool->curr_nr < 4)
				printk("Only␣%d␣elements␣in␣pool!\n", ubi->crypt_cfg->peb_pool->curr_nr);
			if (!tmpbuf) {
				ubi_err("Failed␣to␣allocate␣%d␣bytes␣when"
					"␣reading␣PEB␣%d:%d\n",
					len, pnum, offset);
				return -ENOMEM;
			}

			/*
			 * This is an aligned read and thus won't cause
			 * additional recursions
			 */
			err = ubi_io_crypt_read(ubi, tmpbuf, pnum, aloffset,
						len, 1, 1);
			if (err && err != UBI_IO_BITFLIPS)
				goto out_free;

			memcpy(buf, tmpbuf + offdiff, olen);

out_free:
			mempool_free(tmpbuf, ubi->crypt_cfg->peb_pool);
			return err;
		}
	}
#endif
retry:
	err = ubi->mtd->read(ubi->mtd, addr, len, &read, buf);
	if (err) {
		if (err == -EUCLEAN) {
			/*
			 * -EUCLEAN is reported if there was a bit-flip which
			 * was corrected, so this is harmless.
			 *
			 * We do not report about it here unless debugging is
			 * enabled. A corresponding message will be printed
			 * later, when it is has been scrubbed.
			 */
			dbg_msg("fixable␣bit-flip␣detected␣at␣PEB␣%d", pnum);
			ubi_assert(len == read);
#if defined(CONFIG_MTD_UBI_CRYPTO) || defined(CONFIG_MTD_UBI_CRYPTO_MODULE)
			err = UBI_IO_BITFLIPS;
			goto decrypt;
#endif
			return UBI_IO_BITFLIPS;
		}

		if (read != len && retries++ < UBI_IO_RETRIES) {
			dbg_io("error␣%d␣while␣reading␣%d␣bytes␣from␣PEB␣%d:%d,"
			       "␣read␣only␣%zd␣bytes,␣retry",
			       err, len, pnum, offset, read);
			yield();
			goto retry;
		}

		ubi_err("error␣%d␣while␣reading␣%d␣bytes␣from␣PEB␣%d:%d,␣"
			"read␣%zd␣bytes", err, len, pnum, offset, read);
		ubi_dbg_dump_stack();

		/*
```

```
                 * The driver should never return -EBADMSG if it failed to read
                 * all the requested data. But some buggy drivers might do
                 * this, so we change it to -EIO.
                 */
                if (read != len && err == -EBADMSG) {
                        ubi_assert(0);
                        err = -EIO;
                }
        } else {
                ubi_assert(len == read);
#if defined(CONFIG_MTD_UBI_CRYPTO) || defined(CONFIG_MTD_UBI_CRYPTO_MODULE)
decrypt:
                if (NULL != ubi->crypt_cfg && crypt) {
                        int converted;
                        converted = ubi_crypt_convert(ubi, buf, buf, read,
                                                      pnum, offset, READ);

                        if (converted < read) {
                                ubi_err("ubi_crypt_convert failed, converted %d or %lu bytes\n",
                                        converted, (unsigned long)read);
                                return -EIO;
                        }
                }
#endif

                if (ubi_dbg_is_bitflip()) {
                        dbg_gen("bit-flip (emulated)");
                        err = UBI_IO_BITFLIPS;
                }
        }

        return err;
}
```

D. Oprofile

D.1. Annotated source

ubi_crypt_convert profiling annotations

Listing D.1: ubi_crypt_convert

```
               :int ubi_crypt_convert(const struct ubi_device *ubi, void *buf,
               :                                 void *dst, size_t count,
               :                                 int pnum, int offset, int dir)
    35  0.0454 :{
               :        struct ubi_crypt_result result;
     4  0.0052 :        int i, len, block_size = ubi->hdrs_min_io_size;
               :        int block_num, err = 0;
               :        int nents, tmp_len, rest;
               :        size_t count_save = count;
               :        struct scatterlist *sgentry;
               :        struct sg_table sg_tbl;
               :        struct ablkcipher_request *req;
     5  0.0065 :        struct ubi_crypt_config *ucc = ubi->crypt_cfg;
               :        gfp_t gfp_mask = GFP_NOIO;
    16  0.0208 :        u8 iv[ucc->iv_size];
               :
               :        ubi_assert(is_power_of_2(block_size));
               :
     6  0.0078 :        if (count < block_size) {
               :                printk("ubi_crypt_convert:␣buffer␣must␣be␣larger␣or␣equal␣to␣hdrs_min_io_size,␣"
               :                        "is␣%zu␣but␣should␣be␣at␣least␣%d\n", count, block_size);
               :                return -EINVAL;
               :        }
               :
    18  0.0234 :        len = count > block_size ? block_size : count;
               :
    28  0.0363 :        block_num = ALIGN(offset, block_size) >> ucc->block_shift;
               :        if (ALIGN(offset, block_size) != offset) {
               :                printk("ubi_crypt_convert:␣offset␣must␣be␣aligned␣to␣block␣size");
               :                return -EINVAL;
               :        }
               :
               :        init_completion(&result.restart);
               :
     9  0.0117 :        req = mempool_alloc(ucc->req_pool, GFP_NOIO);
               :
               :        ablkcipher_request_set_tfm(req, ucc->tfm);
               :        ablkcipher_request_set_callback(req, CRYPTO_TFM_REQ_MAY_BACKLOG |
               :                                        CRYPTO_TFM_REQ_MAY_SLEEP,
               :                                        ubi_crypt_async_done, &result);
               :
    63  0.0818 :        while (count) {
               :                err = crypt_iv_essiv_gen(ucc, iv, pnum, block_num);
               :                if (err < 0) {
               :                        printk("Error␣generating␣IV\n");
               :                        goto break_err;
               :                }
               :
    36  0.0467 :                nents = len >> PAGE_SHIFT;
               :                rest = len % PAGE_SIZE;
    31  0.0402 :                if (rest)
    77  0.1000 :                        nents++;
               :
    58  0.0753 :                err = sg_alloc_table(&sg_tbl, nents, gfp_mask);
   117  0.1519 :                if (err < 0) {
               :                        printk("Error␣calling␣sg_alloc_table\n");
               :                        goto break_err;
               :                }
               :
   485  0.6296 :                for_each_sg(sg_tbl.sgl, sgentry, nents, i) {
   258  0.3349 :                        tmp_len = (i == nents - 1 && rest) ?
               :                                                        rest : PAGE_SIZE;
               :
    11  0.0143 :                        sg_set_page(sgentry,
               :                                        mempool_alloc(ucc->page_pool, gfp_mask),
               :                                        tmp_len,
               :                                        0);
               :                }
               :
   101  0.1311 :                sg_copy_from_buffer(sg_tbl.sgl, nents, buf, len);
               :
   192  0.2492 :                err = do_convert(sg_tbl.sgl, sg_tbl.sgl, &iv[0], len,
               :                                     req, dir, &result, ucc->iv_size);
               :
   109  0.1415 :                sg_copy_to_buffer(sg_tbl.sgl, nents, dst, len);
               :
   801  1.0398 :                for_each_sg(sg_tbl.sgl, sgentry, nents, i) {
   181  0.2350 :                        mempool_free(sg_page(sgentry), ucc->page_pool);
               :                }
               :
   202  0.2622 :                sg_free_table(&sg_tbl);
               :
               :                if (err) {
               :                        printk("do_convert error: %d\n", err);
               :                        goto break_err;
               :                }
               :
    52  0.0675 :                count -= len;
    36  0.0467 :                buf += len;
    26  0.0338 :                dst += len;
               :                block_num++;
    62  0.0805 :                len = count > block_size ? block_size : count;
               :        }
               :break_err:
     5  0.0065 :        mempool_free(req, ucc->req_pool);
    35  0.0454 :        return err ? err : count_save - count;
     5  0.0065 :}
               :EXPORT_SYMBOL_GPL(ubi_crypt_convert);
```

D.2. Callgraphs

Callgraph for writing/reading an encrypted block during an iozone benchmark run

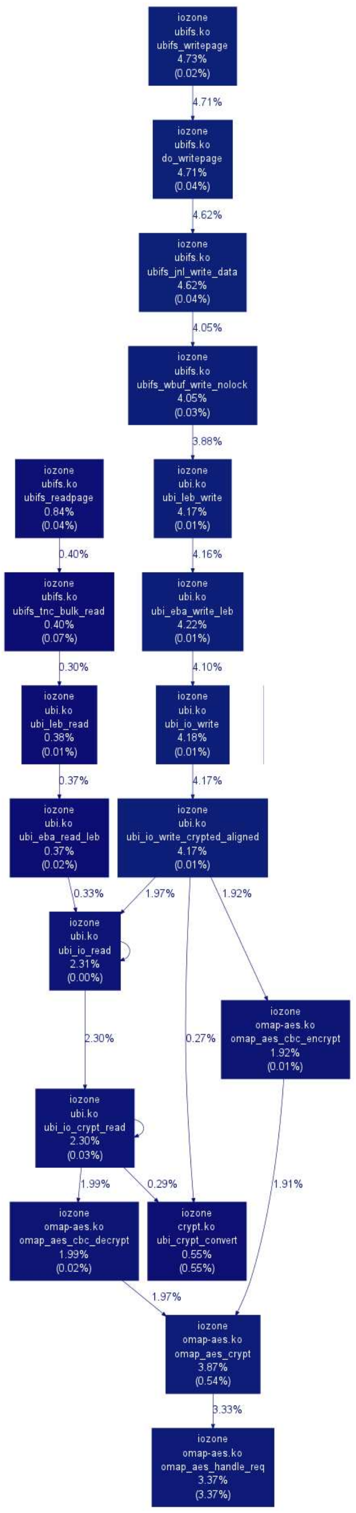


Figure D.1.: Writing/reading from an encrypted UBI device