Digital Investigation 16 (2016) S44–S54

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

DFRWS 2016 Europe — Proceedings of the Third Annual DFRWS Europe

Generic RAID reassembly using block-level entropy

Christian Zoubek [a, *], Sabine Seufert [a], Andreas Dewald [b]

[a] Friedrich-Alexander-University, Erlangen, Germany
[b] ERNW Research GmbH, Heidelberg, Germany

Keywords:
Digital forensics
RAID
Data recovery
Entropy
Tool

A B S T R A C T

RAIDs (Redundant Array of Independent Disks) are widely used in storage systems to prevent data loss in case of hardware defects on a hard disk and to improve I/O performance. In case the RAID controller fails or in the context of a forensic investigation, the content of the RAID has to be reconstructed from the single disks or rather from disk images. Due to the variety of RAID controllers and various implementation and configuration possibilities, different parameters that are necessary for reconstruction are often unknown. This might be the case because the original configuration just has not been documented or in the forensic case, the administrator might not be cooperating and not willing to reveal the configuration. Using the original RAID system in such cases is not an option, too, because the original evidence should not be altered. We present a novel approach to automatically detect all parameters to reassemble the logical RAID volume based on block level entropy measurement and generic heuristics. We also provide a performance-optimized open source implementation of our approach that is also able to afterwards reassemble the entire logical RAID volume and to further recover single missing disks using the redundancy information as present in RAID-5.

Introduction

Redundant Arrays of Independent[1] Disks are a good way to prevent data loss in case of hardware defects on hard disks to a certain degree, while at the same time improving I/O performance (Patterson et al., 1988). However, due to the introduction of another abstraction layer between the hard disks and the operating system, it becomes harder to reconstruct the file system data from the set of disks in case the RAID controller can not be used, as data is distributed among the disks. Due to the variety of hardware and software RAID controllers available and different implementation and configuration possibilities, the RAID type, as well as other important information, such as stripe size and stripe map, may be unknown. Usually, all those parameters need to be determined to allow correct reassembly of the RAID volume (Arpaci-Dusseau and Arpaci-Dusseau, 2012; Katz, 2010). If some of the parameters are not known, raw disk contents have to be manually examined by specialized analysts to determine the right parameters, making RAID reassembly a complex and expensive task today (Xiang et al., 2010).

Motivation

In the field of forensic computing (also called digital forensics), where accessing data on previously seized and imaged hard disks is the basis for many investigations, RAID reassembly is specifically important in analysis of server environments, most of which use RAID systems in order to guarantee failure safety. To allow for evidence collection, the RAID volume has to be reconstructed from the single disk images, where the original RAID system can not be used (to not alter the evidence), and where the

* Corresponding author.
E-mail address: christian.zoubek@fau.de (C. Zoubek).

[1] Originally “Inexpensive” but not common today.

http://dx.doi.org/10.1016/j.diin.2016.01.007

relevant parameters are also often unknown. This might be the case because the original configuration has not been documented after its installation or because the administrator/owner might not be cooperating and thus be not willing to reveal the configuration. As such constellations occur regularly in today's investigations, forensic tools are needed that allow for automated reconstruction of the RAID. Common forensic tools already support reassembling the logical RAID volume from single disk images, but the investigator has to know all the parameters and provide them to the software. If the configuration is not known, the only remaining option is the already mentioned manual recovery, which is often too time-consuming and expensive. Thus in 2012, in his talk on the DEF CON 20 conference, Michael Perklin even suggested using custom RAID parameters as an effective anti-forensics measure (Perklin, 2012).

Contributions

We present a novel approach to automatically detect all the relevant RAID parameters in a generic way using block level entropy measurement and generic heuristics. We then use these parameters to reassemble the original RAID volume. More specifically, we employ a heuristic based on entropy patterns of files that allow us to find possible locations of stripe boundaries and potential points of cohesion across the different hard disks. This way, we gather indications for possible RAID types and parameters from which we detect the original configuration. We provide an open source prototype[2] that is able to detect the following parameters of RAID-0, RAID-1 and RAID-5:

1. The RAID type
2. Stripe size
3. Stripe map and Disk order

Our prototype is also able to automatically reconstruct a single missing hard disk in case of a RAID-5 system using the parity information of the RAID and to reassemble the complete logical RAID volume as a single disk image. Using our implementation, we evaluate the performance and correctness of our proposed method.

Related work

Entropy-based heuristics are already used in some scenarios that are quite different from our approach: The work of Xiong et al. (2014) introduces an approach of using entropy to evaluate risk factors for certain movement patterns acquired from mobile device users. That way, conclusions about the security of mobile devices can be drawn by examining entropy patterns. In geographical regions where the user moves most of the time, the entropy is very low and so is the risk factor. This allows for risk-based authentication. The work described in Yokota et al. (2007) shows the use of entropy of control flow branching histories to quantitatively describe the regularity level of program behavior to improve branch prediction performance.

Garfinkel et al. (2010) describe the use of purpose-built functions and cryptographic hashes in the context of small block forensics. Their work has two different parts: Block hash calculations and bulk data analysis. The hash allows to detect compressed data and similarity and they use it to improve detection of JPEG, MPEG and compressed data for classifying forensic contents of a drive in case of data carving. A possible extention of our work could be to use their hash additionally or instead of entropy in order to detect the RAID parameters in future work.

Xiang (2011) describes an approach to recover data from a RAID-6 system. In contrast to the RAID systems used in this paper, RAID-6 provides higher level of reliability through two parity stripes. RAID-6 codes like RDP protect data against up to two disk failures (Corbett et al., 2004). Once a single disk failure is detected on the running RAID-6 system, recovery is initiated without aborting application activity on the system. This approach attempts to reduce the number of reads required for recovery to minimize the performance impact for other applications. In contrast to our approach, this approach works on a running system and therefore with knowledge over all RAID parameters.

Kiselev et al. (2006) describe an approach to automatically detect corrupt data in a parity RAID system and provide recovery. With two different parity data they can determine, if data is corrupt and restore the original data.

Belhadj et al. (2003) describe how the rebuilding of data on a RAID system can be prioritized using a vulnerable data redundancy scheme. This scheme can be determined by comparing probability of losing data and the potential for one or more disk failures.

The work presented by Hart (2002) describes an automatic approach to restore the configuration of a RAID system after a system failure, when the operating system is running on that RAID system.

The algorithm described by Goel and Corbett protects from three-disk-failures (Goel and Corbett, 2012) using three parity disks in respect to three recoverable disks.

We further want to name two existing (non-academic) tools for RAID reconstruction: *RAID Reconstructor* (*v 4.32*)[3] and *ReclaiMe* (*build 1994*).[4] In our evaluation, we compare both tools to our approach.

Outline

This paper is structured as follows: In Prerequisites, we provide the theoretical prerequisites for this work and recall the specifications of different RAID types and their parameters, as well as entropy. Parameter detection using entropy describes our approach of entropy-based parameter detection in detail. We evaluate the correctness of our approach, as well as its performance in Evaluation and conclude in Summary.

[2] https://www1.informatik.uni-erlangen.de/content/forensic-raid-recovery/.

[3] https://www.runtime.org/raid.htm.

[4] http://www.reclaime.com/library/how-to-recover-raid.aspx.

Prerequisites

In this section, we recall the important basics for our approach. To this end, we first outline the functional principle of RAIDs, the different RAID types, and the parameters that are relevant for reassembling the volume. We then also briefly review the definition of entropy and provide some examples to give a first intuition of why entropy can be used to obtain the information about the previously mentioned RAID parameters.

RAID types and parameters

RAID is a means of unifying multiple physical hard disk drives to a single logical volume. The main idea of RAID systems is to achieve redundancy which is used to guarantee additional safety in case of hard disk failures and increase I/O performance (Chen et al., 1994). These advantages make RAID systems especially viable for server environments. However, the technology is also applied by private individuals in small network-attached storage (NAS) solutions. As aforementioned, RAID systems operate transparently to the file system and appear as one logical device. This is achieved by having a so-called RAID controller to perform read/write operations on the hard disks in their respective, redundancy-aware manner. Said redundancy allows for a complete recovery of the contents of a broken hard disk drive without losing data. Depending on the RAID types used, which we revisit now, the required recovery procedures differ in complexity and error tolerance.

Most RAID types use striping across the single hard disk devices to increase read and write performance. *Striping* refers to distributing data blocks (so called *stripes* of a specified *stripe size*) sequentially over the devices firstly and over physical addresses secondly, so that data on a disk is not contiguous. The resulting *stripe map* describes the mapping of logical data blocks to their physical location in the array of disks. This location consists of the device a given data block is located on and its address (offset) on that device. Depending on the RAID controller, stripe sizes may be chosen. The smallest possible stripe size is the size of a hard disk sector (512 Byte), yet there is no upper limit defined.

The most basic example for striping is RAID-0, where the stripe map describes a linearization of the data blocks over the devices, as illustrated in Fig. 1. Since data can be written and read in parallel, the I/O performance is sped up, ideally by a factor equal to the number of devices. However, any complete hard drive failure results in unrecoverable and non contiguous data and therefore in complete data loss. The stripe map only consists of the disk order.

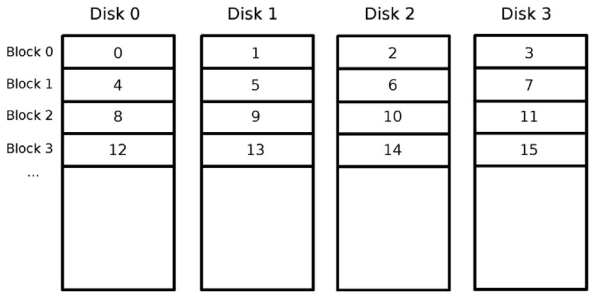


Fig. 1. Example for data block distribution of a RAID-0 system with four disks.

While RAID-0 provides striping and no redundancy, RAID-1 poses an alternative without striping but redundancy, where the logical drive is mirrored across physical drives. Any number of hard disk drives may fail, as long as one remains intact, and recovery is trivial. Since no striping occurs, there is a performance gain by reading data, but a bottleneck imposed on writing by the slowest writing device. RAID-1 only leaves $1/n$ of the system's accumulated space available, for n equally sized hard disks.

All other RAID types provide redundancy via parity bits, which are usually realized by a bit-wise XOR. The approach presented in this work does not provide support for RAID-2 through RAID-4 types, which have been succeeded by RAID-5 and are thus rarely used in the field.

Along with RAID-0 and RAID-1, RAID-5 is the most commonly used RAID type. It combines redundancy through parity and performance through striping, thus three hard disks are at least required. There are different types of parity and data block distribution, as explained in great detail by Lee (1990). In general, the resulting stripe map of the distribution is repeated after n rows, with n being the number of hard disks. Those distribution types are illustrated for 4 disks in Table 1. There are two properties with two possible states each, namely {left, right} and {symmetric, asymmetric}. Left-sided distributions shift their parity blocks from the last disk to the first, while right-sided ones arrange them in a first-to-last manner. Symmetry refers to starting the row-wise data sequence after the parity block, whereas asymmetric placement begins each row on the first available disk of that row (Lee, 1990).

Table 1
Parity and data block distributions for Raid-5.

Disk 0	Disk 1	Disk 2	Disk 3
(a) Left-asymmetric			
0	1	2	P
3	4	P	5
6	P	7	8
P	9	10	11
(b) Right-asymmetric			
P	0	1	2
3	P	4	5
6	7	P	8
9	10	11	P
(c) Left-symmetric			
0	1	2	P
4	5	P	3
8	P	6	7
P	9	10	11
(d) Right-symmetric			
P	0	1	2
5	P	3	4
7	8	P	6
9	10	11	P

The storage efficiency of RAID-5 is greater than in RAID-1, because instead of using $1/n_n^1$ of the system's accumulated space for data, only $1/n_n^1$ of the space is used for parity and the remaining disk space is available for data storage. If all data is to be restored without loss of information, a maximum of one hard disk failure is tolerated. By spreading the parity information over multiple devices, the load is equally balanced and the performance of RAID-5 behaves just like its storage efficiency for the same reason: All disks but one contain actual payload.

All those parameters, namely RAID type, stripe size, stripe map and of course the disk order need to be determined to be able to reassemble a RAID volume. Determining all of those requires a heuristic approach, since testing all possibilities for consistency (i.e. brute forcing) is not viable as we show in Bruteforce performance estimation. The underlying concept of the heuristics introduced in this work is informational entropy, which we want to explain next.

Informational entropy

Entropy essentially is the expectancy value of information content. For a discrete probability distribution $(p_1,p_2,\ldots)$ entropy is defined by Eq. (1), as also mentioned by Devroye and Luc (1996).

$$H = -\sum_i p_i \times \log(p_i) \tag{1}$$

This means, entropy describes how much different information of length n can be found in a given sequence of data. For example, let the sequence of data be a stream of eight consecutive logical data units, and let the length of information n be two. Each single data unit can take the values 0 and 1. In consequence, information of the length two can only be described by four possible values, namely 00, 01, 10 or 11. If the stream only consists of one kind of value, information of only this type is encountered and the entropy is zero (Shannon, 2001). The greatest entropy is always achieved if all possible values of information occur equally often. As shown in Table 2, the highest entropy for this example is two. The first table shows exemplarily the results if only one type of information is present. The second table shows an example for maximum entropy in this simple example, where all possible values occur equally often.

Table 2
Example for relation between information content and entropy.

	#	p	$p \times \log_2(p)$
00	4	1.0	0
01	0	0.0	0
10	0	0.0	0
11	0	0.0	0
$H = -\sum_i p_i \times \log_2(p_i) = 0$			
00	1	0.25	−0.5
01	1	0.25	−0.5
10	1	0.25	−0.5
11	1	0.25	−0.5
$H = -\sum_i p_i \times \log_2(p_i) = 2$			

Generally, the lowest entropy is zero, while the highest entropy is n. Since text files have a limited range of values (because only printable characters are used, and for example the letter 'e' is very common, while special characters like '#' are rarely used) the entropies of blocks over a whole file are rather low (Shannon, 2001). To get higher entropies, the content of a file has to be fairly random with many different values. For example, compressed files have a higher entropy, since redundancy is eliminated and the randomness is increased. The other way round, in a completely random file, nothing can be compressed (Salomon and Motta, 2010). As a result, the entropy of any block of an compressed file or a file that contains arbitrary binary data is rather high, as well as the entropy of the entire file.

We make use of this fact to detect data boundaries as explained in the next section. The algorithms described there calculate the entropy with the information length of one byte. Thus we have 2^8 possibilities for each byte to differ, the highest entropy for each block is $\log_2(2^8) = 8$, and the size of a block has to be a multiple of 2^8.

Parameter detection using entropy

In order to reconstruct a logical RAID volume, all the previously explained parameters of the RAID system have to be known. We detect those parameters in the following order, as they partly depend on each other:

1. RAID type
2. Stripe size
3. Stripe map
4. Data offset

All of those can be determined using heuristics on block level entropy of the disk contents. Those different heuristics for each of the parameters are explained in the given order in the following sections.

RAID type

As described in RAID types and parameters, the data blocks (stripes) in a RAID can be mirrored over different disks or striped with or without parity information. In order to detect which RAID type has originally been used, occurrences of mirrored or parity blocks are counted, as well as blocks that cannot be assigned. The RAID type can then be determined as seen in Table 3 on the following page. In this table, RAID-5i denotes an incomplete RAID-5 system with missing disks, whereas RAID-5c is a complete one.

Table 3
Overview of the relation between the number of found blocks with the respective property and the determined RAID type.

	RAID-0	RAID-1	RAID-5c	RAID-5i
mirrored	low	high	low	mean
parity	low	low	high	mean
unassigned	high	low	low	high

In order to count how many blocks are either mirrored, contain parity information, or can not be assigned, we proceed as follows: The algorithm examines the first 5 million non-zero 512-Byte (smallest possible stripe size) blocks of each disk. Using a bit-wise XOR, the algorithm can determine if a block is mirrored onto another disk (XOR results in 0-bytes only) or if a block with parity information has been found. Using this information, the RAID type can be derived in the following way:

As RAID-1 has mirrored disks, the XOR of any two blocks evaluate to zero. Therefore, most examined blocks have to be mirrored if all disks are correct. RAID-5 uses parity blocks for failure safety which are also calculated with a XOR operation. For this reason, in a RAID-5, the XOR of a block of *all* disks is calculated, it should evaluate to zero. This way, the algorithm finds many parity blocks on a complete RAID-5 system.

Since RAID-5 can be recovered with one missing disk, the algorithm has to detect the difference between a RAID-0 and an incomplete RAID-5 system. Both RAID systems have a lot of blocks that are neither mirrored nor have redundant information. Still, if the missing disk of a RAID-5 system has empty blocks, then mirrored and parity blocks can be found on the remaining system. The heuristic respects this case and expects a certain number of unassigned blocks, as can be seen in Eq. (2), where n denotes the number of disks:

$$mirrored < \frac{100 \times unassigned}{n+1} \tag{2}$$

However, if the number of mirrored blocks and blocks with redundant information is too low, than those occurrences could be coincidences. That's why both values summed up should be a fraction of the number of unassigned blocks. Those two conditions are sufficient to distinguish an incomplete RAID-5 system from a RAID-0 system.

Stripe size

The length of information for calculating the entropy of a block is chosen to be one byte. With one byte having $2^8 = 256$ possible values, the block size has to be at least 256 byte. If the length was chosen to be two byte, the blocks to determine the stripe size have to be at least 512-byte blocks. This is already the smallest possible stripe size and thus we can not increase it further without loosing compatibility with this stripe size. To decrease the length to less than one byte makes no sense because one byte is typically the smallest data unit used. Thus, the only two remaining possibilities for block sizes that can be chosen for entropy calculation in our approach are 256 and 512 byte blocks and one or two byte information length. We chose 512-byte blocks with one byte information length as basis for our calculations, as this showed to decrease the impact of outliers in entropy and thus provided the best results.

As mentioned in Prerequisites, the entropies within a certain file remain roughly the same. Thus, great differences in entropy of consecutive blocks may indicate the beginning of a new stripe. An example for an entropy distribution on a RAID-0 system with image files is shown in Table 4:

The high entropy shows the content of the file. It is surrounded by empty blocks. The possible stripe size can be derived using the addresses of the blocks with a high difference in entropy, i.e. the file boundaries. In this example, address 888274944 indicates the beginning of a stripe. The entropy on Disk 1 to Disk 3 have a lot of blocks with low entropy before, until the value changes abruptly to ≈ 7.5 while the entropy on Disk 0 remains zero. This difference in entropy is unusual for a consecutive written file as the entropy should not change severely in such a case. It is also untypical for the beginning of a file, thus it can be interpreted as the beginning of a stripe.

The end of a stripe can be located at address 888733184, since there is also an abrupt change of value on Disk 3. This discontinuity to the following 512-byte block is unlikely found within a file and at the end of a file, the entropy usually slightly drops before it reaches zero (at least when the end of the file is not exactly on a stripe border), similarly to the beginning of the file. As a consequence, a new stripe block starts at address 888733696, where the file continues on disks 0 to 2. To ensure that a strongly altering file with partly low entropy does not influence this approach, the surrounding entropies are also taken into account. In case of an abrupt change of entropy from very low to very high, the values before the change have to stay at the same low level, whereas after the change, the values have to stay higher than a certain threshold. The best threshold may vary depending on the examined case, which is evaluated and explained in Evaluation. The presented algorithm checks sixteen 512-byte blocks of entropies before and after the entropy change. If the entropy changes from high to low, this procedure is also applied the other way round.

To finally estimate the possible stripe size, the difference between two consecutive addresses is calculated. The result has to be a multiple of the possible stripe size and is classified using modulo operations of reasonable stripe sizes. The largest applicable divisor of the modulo operations, where the remainder is zero, is taken as a potential

Table 4
Entropy distribution of a 1.75 MB file over four disks using RAID-0.

Address	Disk 0	Disk 1	Disk 2	Disk 3
…				
888273920	0	0	0	0
888274432	0	0	0	0
888274944	0	7.50199	7.56131	7.57583
888275456	0	7.53411	7.54758	7.54145
…				
888306176	0	7.46816	7.43265	7.48876
888306688	0	7.43318	7.59278	7.60496
888307200	6.14066	7.48741	7.58424	7.49408
888307712	7.64113	7.53735	7.59764	7.46034
…				
888732672	7.43689	7.55090	7.52364	7.54029
888733184	7.52416	7.54816	7.57045	7.53455
888733696	7.44034	7.54581	7.46290	0
888734208	7.47576	7.51771	7.57273	0
…				

stripe size. This is calculated for the entire discs and in the end, the potential stripe size that occurred for most addresses is assumed to be the original stripe size. (In fact, usually the figures are very unambiguous, as we show in our evaluation in Evaluation.)

Possible offset

Since Software-RAID systems need reserved space for their own metadata, an offset to the actual RAID system must be known. In case of RAID systems created with the tool *mdadm* (version 3.2.5), a superblock with details about the configuration of the RAID system is stored 4 KB after start of each device. Depending on the disk layout, the RAID system and the type of the Software-RAID, the offset to the actual RAID system may vary.

The algorithm to determine the stripe size, as already discussed in Stripe size, was extended to provide additional offset detection. After the stripe size has been determined, the difference between two addresses, that have lead to the stripe size determination, is calculated. The result of the difference has to be a multiple of the stripe size. That way it can be assumed that these addresses indicate a correct stripe withing the RAID system. Since those addresses are most certainly within the file data of the RAID system, the lower address of the calculated difference is taken as the maximal possible offset. This offset is then taken modulo with the stripe size and the result is the lower bound for the offset algorithm. That way the maximal range is determined, in which the striping of the RAID system should begin.

Between this range a search algorithm is applied to find magic values at certain offsets for different file system metadata structures. In case of a partition table, the magic value 0x55aa is found at offset 0x01fe (decimal: 510). Also file systems like NTFS with NTFS as magic value, or Ext with 0xef53 as magic number, are supported. The smallest possible offset, which can be detected using these magic values, is taken as the Software-RAID offset and is used on all disks.

Stripe map

After determining the stripe size and a possible offset, the stripe map has to be reconstructed for RAID-0 and RAID-5 systems. For this purpose, we make use of the fact that the striped data blocks are written consecutively across the hard disks. As a result, empty data blocks can deliver a hint if this block was written before or after another non-empty stripe block. This depends on the entropy distribution of surrounding blocks. We divide each stripe into two parts and calculate the entropy for each part individually to detect falling and rising edges in entropy within a stripe. Falling edges may be an indicator for the end of a file, whereas a rising edge indicates the beginning of a file.

If such an edge is found, the address is considered to be of interest. Furthermore, the data blocks of those *border candidate* addresses on the other disks are examined. An empty block has to be written before the data block with a rising edge or after a data block with a falling edge. That way, the disk order can be derived and more importantly the write order of the data blocks. As mentioned in Prerequisites for a RAID-0 system, only the disk order is required.

In case of a RAID-5 system, there are different rows of the stripe map that have to be considered, as well as the parity block distribution that has to be determined, as already explained in Prerequisites. The border candidate addresses are used to calculate the position within the stripe map, as shown in Eq. (3). Let n be the number of hard disks of the RAID system and s be the determined stripe size. The row of the stripe map r is then calculated using the border candidate address a in the following way:

$$r = \left(\frac{a}{s}\right) \bmod (n) \tag{3}$$

To determine the parity distribution for RAID-5 systems, non-zero data blocks are examined, again using entropy. Since parity blocks are calculated by XOR, their content is usually more random than the content of the corresponding data blocks (for non-empty blocks). Those found parity blocks are counted and are associated with an entry in the stripe map. Using Eq. (3), the position within the stripe map is calculated. The final distribution of parity blocks is quite reliable, so that it can be set with certainty. Examples for bound cases of possible parity block distributions are shown in Evaluation.

The described information is used to determine the type of the RAID-5 system. As mentioned in Prerequisites, a RAID-5 system can be {left, right} and {asymmetric, symmetric}. For this purpose, the parity distribution provides an initial situation. Afterwards, the type of the RAID-5 system can be determined by applying the following simplified algorithm:

```
1  curNum = -1
2  for (i = 1; i < #rows) {
3    for (j = 0; j < #cols) {
4      if (stripeMap[i][j].isParity()) {
5        if (stripeMap[i-1][j].blockNum.isValid()) {
6          if (curNum == -1) {
7            curNum = stripeMap[i-1][j].blockNum
8          } else {
9            if (stripeMap[i-1][j].blockNum > curNum)
                 {
10             asymm_right = true
11           } else if (stripeMap[i-1][j].blockNum <
                 curNum) {
12             asymm_left = true
13           } else {
14             if (stripeMap[i-1][j].blockNum == 0) {
15               symm_right = true
16             } else {
17               symm_left = true
18 } } } } } } }
```

For each row in the stripe map, the hard disk with the parity block is located. Then, the previously written data block on that hard disk is examined. In every row, this data block has to be the first block or the last block to be written

for a right symmetric or respectively a left symmetric RAID-5 system. In the asymmetric case, these data blocks have to be in a ascending or descending order – respective to the rows in the stripe map – to indicate a right or a left RAID-5 system. These patterns can also be seen in Table 1 on page 3 in Prerequisites.

Evaluation

In this section, our approach is evaluated using our implementation. For the different kinds of RAID types and for different data sets, the stability, performance and correctness was tested.

Data set

In order to evaluate our approach on a realistic data set, we used the hardware RAID controller the Adaptec®6405 Family Controller. To cover different RAID types, as well as different stripe sizes and file systems, we combined each of those with each other. In addition, for each of those cases, we generated two variants with the most problematic border cases regarding entropy measurement: Disks with low-entropy data (text files) and high entropy data (compressed picture files, i.e. JPEGs).

- RAID types: RAID-0, RAID-5 (as discussed before, RAID-1 is trivial)
- Stripesizes: 16 KB, 64 KB, 256 KB, 1024 KB
- File system: Ext4 and NTFS
- Data: Picture files and crawled text, and picture files

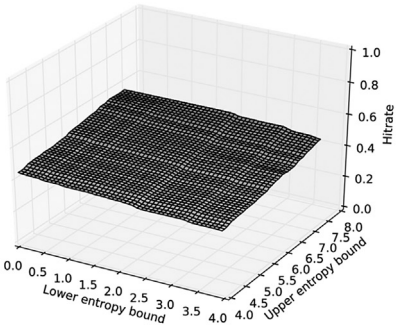
Thus, we obtained $2 \times 4 \times 2 \times 2 = 32$ RAID systems with 4 disks each of 10 GB size and additionally three 500 GB Ubuntu Linux system disks for RAID-0 and for RAID-5, which makes up 38 RAID systems with a total of 152 hard disk images on which we evaluated our proposed method. Furthermore several RAID-5 systems were created using the tool mdadm to evaluate the correctness of the offset detection for Software-RAIDs.

In order to provide a data set that not only covers the potentially most problematic cases, but also is as close as possible to real world images, the images with an Ubuntu Linux installation have further been in use to ensure that data was also moved, allocated and deallocated on the image.

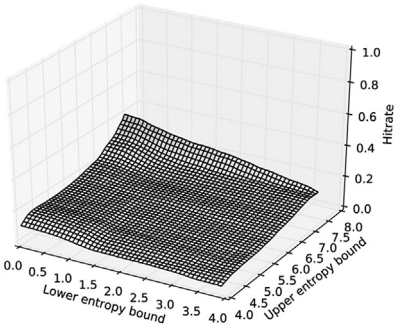
Determining optimal thresholds

As already mentioned in Parameter detection using entropy, certain thresholds for the differences in entropy need to be set to find the stripe size. For different data sets and file systems, the optimal thresholds differ. Still, there exist multiple threshold combinations for every data set with which the recovery was successful. Calculating intersections of these combinations, we determined a universal threshold. Alternatively, thresholds could be set for the specific case, if there is enough prior knowledge of the characteristics of the files and the file system, but our evaluation shows that this is not necessary.

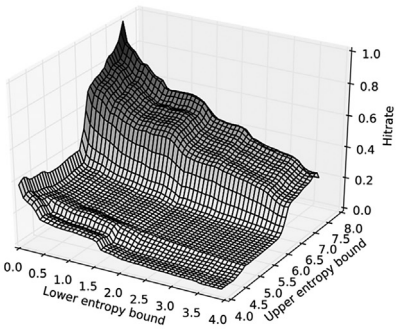
Fig. 2 on page 11 shows some selected examples for the hit rate for the correct stripe size (y-axis) depending on the chosen different upper and lower thresholds (x- and z-axis), for detecting the stripe size. Subfigure 2a shows that the combination of NTFS and large picture files is greatly independent of the chosen thresholds and is very robust. Since there only occur high entropies from the picture files



(a) Distribution for NTFS filled with picture files using RAID-5 with 256KB stripe size



(b) Distribution for Linux operating system using RAID-5 with 512KB stripe size.



(c) Distribution for Ext4 file system filled with many small files using RAID-0 with 1024KB stripe size.

Fig. 2. Distribution of stripe size algorithm reliability using certain entropy threshold combinations. The higher the value on the z-axis, the better the reliability. The other axes show the tested entropies for upper and lower thresholds.

($\approx$7.9)and very low entropies from empty blocks ($\approx$0.0) – which are the edge cases – the chosen thresholds do not cause different results. The other two distributions, shown in Fig. 2b and c, show a difference in the results. The peak of both graphs is achieved with a high distance between both thresholds. Many different types of files, as exemplarily encountered in operating systems, lead to highly varying entropies. Hence, consecutive stripe blocks are more difficult to be told apart. As a consequence, the thresholds must be chosen more carefully to not misinterpret an entropy change within a file as the beginning of a new stripe block. In the end, the best lower and upper thresholds are estimated depending on those results. For all cases of the data set described in Data set the most fitting thresholds are 0.3 for the lower threshold and 7.3 for the upper one, in a way, that in the average the best results can be achieved using these thresholds.

Correctness

As described in Parameter detection using entropy, the border candidate addresses are used to calculate potential stripe sizes. Fig. 3 shows potential stripe size distributions using the previously determined thresholds with the y-axis being the probability of the stripe sizes. For all graphs, the result is obvious and the correct stripe size can be determined.

As argued in Parameter detection using entropy, parity blocks of a stripe should have the greatest entropy within a row. We verified this assumption using our data set and the results shown in Table 5 on the following page, which displays the found parity distributions for different RAID-5 stripe maps. Table 5a shows an explicit parity distribution, since the large picture files have uniform entropy values. In contrary to Table 5b, the picture files may be greater than the size of a stripe, what makes great changes in entropy unlikely. A lot of small files can cause high changes in entropy inbetween a stripe block. Although in this case, shown in Table 5b, the parity map is not that obvious, it still can be accurately derived.

In case of a RAID-0 system, the parity distribution can not be determined, since this RAID type does not support parity information. The calculation of the resulting stripe map depends only on the disk order.

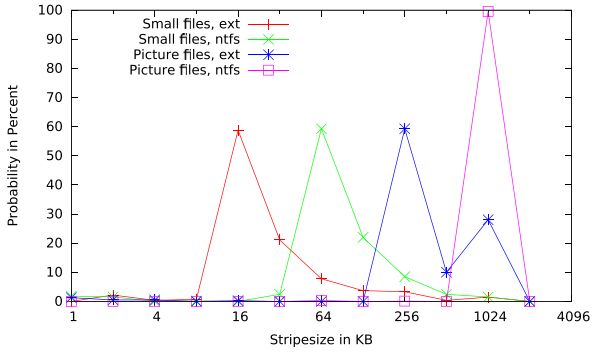


Fig. 3. Probability distribution of potential stripe sizes for certain cases. The peaks indicate the stripe size for the specific case.

Table 5
Parity distributions of the stripe map for different RAID-5 setups.

Disk 0	Disk 1	Disk 2	Disk 3
(a) Parity distribution for picture files			
0	4958	0	0
0	0	5002	0
0	0	0	4911
4922	0	0	0
(b) Parity distribution for small files			
485	480	497	3805
469	512	3808	478
499	3785	490	498
3800	518	442	510

We applied our implementation on all the RAID systems from our previously described data set and let it reconstruct the logical RAID volume. We verified the correct reassembly by mounting the resulting images and analyzing both the partition tables and file system metadata.

For all cases in the data set but one, our tool could successful detect all necessary parameters automatically and reassemble the original RAID volume. The only not recoverable case is the RAID-0 system with Ext4 as a file system, a lot of small files and a large stripe size. This combination of a large stripe size with very small files on RAID-0, where no parity information is present, is a problematic case, since there is not enough information provided to successfully recover the correct disk order automatically. In such a case, the investigator has to provide the original disk order if known, or try out different orders, which is $n!$ and for many real cases with 3 disks (at most 6 tries) or maybe even 4 disks (24 tries) still might be reasonable. Yet the algorithm may already find some part of the order.

Performance

Finally, we evaluated the runtime performance of our prototype, to underline the practical applicability of our approach and to demonstrate that the overhead for calculating the entropy values in order to automatically detect the needed RAID parameters does not render the approach useless. Measurements were taken with the following setup:

- CPU: Intel©i5-760; 4 Cores @ 2.8 GHz
- 8 GB RAM @1.333 GHz
- Disks connected via SATA-2

Detecting all parameters of a RAID system with 40 GB needs $\approx$4.5 minutes for best cases and up to 8.5 min for worse scenarios. The greatest decrease in performance occur if disks are barely filled with data. In this case, for each parameter to detect the disks have to be fully searched to find enough indications.

For the recovery of lost disks, the write performance reached an average of 65 MB/s. The reassembly of the whole system (without a missing disk, or after it has been recovered) reached a write performance of 138 MB/s. This

means, it takes about 12 min and 22 s for recovery of 100 GB if all disks are present.

Bruteforce performance estimation

In those rare cases where a custom configuration has been used, where no parameters are known, and the automated detection fails, the only remaining alternative approach is to bruteforce the parameters. In the worst case, each possible combination of the parameters would have to be tested. For each tried combination, at least a part of the image has to be recovered to run some consistency tests to detect, whether the right parameters have been chosen or not. Depending on the data on the image and the file system used this is not a trivial task, but we want at least to give some hints. One possible approach would be to detect the file system used, then guess in which part of the entire image the file system metadata is usually stored to reassemble this part (respecting offsets) and then try to interpret the file system structures to obtain a listing of the root directory, for example. Even if this succeeds this does not ensure that everything has been assembled right, because all the information might fit in a single or only few consecutive stripes, so that a partially correct reconstruction would be enough to pass this check. However, such a test could reveal possible candidates of correct configuration guesses, which can then be used to fully reassemble the logical volume under the assumption that the configuration is correct and manually search for the correct image among those candidates.

To give an intuition of the effort that has to be done in such a case, we want to show how to calculate the number of possible configurations, using our data set as an example. Let then s denote the possible values of the stripe size, r be the number of possible RAID types except RAID-1 and m be the possible configurations of the stripe map. Since RAID-1 has only one configuration, this case must be counted on top. Then N is the resulting number of configurations which have to be tested:

$$N = (s \times m \times r) + 1 \tag{4}$$

Here, s can take ten different values since the stripe size can be set from 1 KB to 1024 KB. The RAID type r can be RAID-0 and RAID-5 which makes two options (since RAID-1 is trivial and does not need reconstruction if at least one image is given). Generally, the stripe map configuration depends on the distribution of the blocks in one row across the disks, which makes $P!$ possibilities for P being the number of devices. If the four different configurations of the stripe map are taken into account, as they are described in Sec. 2 and for example four devices, as in our evaluation data set, m would be calculated as $P! \times 4 = 96$. Thus, we would end up with $N = (10 \times 96 \times 2) + 1 = 1,921$ possible configurations for our example RAID systems.

Comparison to other tools

We now want to draw a short comparison of our implementation with the two already mentioned existing tools *RAID Reconstructor* and *ReclaiMe*.

RAID Reconstructor cannot detect RAID types automatically, but it is able to find the start sector. It uses differentiate entropy to analyze the RAID to this end. Yet the entropy is mostly too low to determine any parameter and the tool aborts with the message “This result is not significant”. Unfortunately, RAID Reconstructor fails for *all* the images in our previously described test dataset.

ReclaiMe lacks the ability to automatically detect RAID types, but, compared to our tool, it additionally supports the detection of parameters for nested RAID arrays. ReclaiMe is able to handle most of the images in our data set but in some cases is unable to find the necessary parameters, detects wrong parameters and thus fails, or exits with a “blank Disk error” (the tool assumes the disk images are empty, which is not the case). The detailed evaluation results of ReclaiMe for our data set are shown in Table 6 for the RAID 0 images, of which 52.6% are reconstructed correctly, and in Table 7 for the RAID 5 images, of which only 15.8% are correct. Especially, for RAID systems with large stripesizes (e.g. 1024 KB) can hardly be recovered. In the cases for which the parameter detection works, it takes around 23 min for the 40 GB RAID systems on our previously described test hardware setup compared to the 4.5–8.5 min that our tool takes for the detection. Thus our tool provides better results (detects all but one test cases correctly) and is even much faster than ReclaiMe.

Summary

In this work, we presented a novel approach to automatically detect relevant RAID parameters in a generic way using block level entropy to reassemble the original RAID volume. By using entropy, the presented algorithm is able to find possible locations of stripe boundaries and potential points of cohesion across the different hard disks to estimate the stripe size and the stripe map of the RAID system. We provide a ready-to-use implementation for the

Table 6
Evaluation results of *ReclaiMe* for RAID 0.

FS	Stripe S.	Content	Cor.	Incor.	not found	Error
Ext	512	Ubuntu				x
Ext	128	Ubuntu				x
Ext	32	Ubuntu				x
NTFS	16	Pictures	x			
NTFS	64	Pictures	x			
NTFS	256	Pictures	x			
NTFS	1024	Pictures			x	
Ext	16	Pictures			x	
Ext	64	Pictures			x	
Ext	256	Pictures			x	
Ext	1024	Pictures			x	
NTFS	16	Crawl	x			
NTFS	64	Crawl	x			
NTFS	256	Crawl	x			
NTFS	1024	Crawl	x			
Ext	16	Crawl	x			
Ext	64	Crawl	x			
Ext	256	Crawl	x			
Ext	1024	Crawl			x	
Sum			**10**	**0**	**6**	**3**
%			**52.6**	**0**	**31.6**	**15.8**

The bold text are the sums of the upper values.

Table 7
Evaluation results of *ReclaiMe* for RAID 5.

FS	Stripe S.	Content	Cor.	Incor.	not found	Error
Ext	512	Ubuntu				x
Ext	128	Ubuntu				x
Ext	32	Ubuntu				x
NTFS	16	Pictures		x		
NTFS	64	Pictures		x		
NTFS	256	Pictures		x		
NTFS	1024	Pictures		x		
Ext	16	Pictures			x	
Ext	64	Pictures			x	
Ext	256	Pictures			x	
Ext	1024	Pictures			x	
NTFS	16	Crawl		x		
NTFS	64	Crawl		x		
NTFS	256	Crawl		x		
NTFS	1024	Crawl		x		
Ext	16	Crawl	x			
Ext	64	Crawl	x			
Ext	256	Crawl	x			
Ext	1024	Crawl			x	
Sum			**3**	**8**	**5**	**3**
%			**15.8**	**42.1**	**26.3**	**15.8**

The bold text are the sums of the upper values.

presented algorithm with high performance for reasonably sized RAID systems that is able to handle RAID-0, RAID-1 and RAID-5.

Conclusion and limitations

We conclude that the basic principle of reassembling RAID systems with entropy based heuristics can be applied with suitable thresholds and parameters. Our tests show that, in most of the cases, a complete reassembly of the RAID system including the automatic detection of all of the RAID parameters was possible in adequate time. Our software should be extensible in such a way as to support RAID-3, RAID-4 and RAID-6 systems due to their conceptual similarity to RAID-5. Due to the prevalence of common parity distribution patterns of RAID-5 systems, the algorithm is able to reassemble the RAID system regardless of hard disk order.

However, the lack of parity information in RAID-0 systems makes the problem of determining the hard disk order difficult. Nevertheless, the types of encountered files are of great relevance to the accuracy of finding the parameters. Large picture files are most beneficial, while small files of any type provide less information through entropy analysis.

In case empty space on disks would not be zero-filled, but initialized with random data (that typically has high entropy), our approach would not work. Luckily, all current RAID systems that we are aware of initialize the disks with zeros only. Similarly, in case of encrypted file systems on the RAID and fully encrypted images, which have a (uniquely) high entropy across the entire image, the correct parametrization can not be detected using our approach. In such cases our approach does not help in reconstructing the RAID volume, but in such cases we have to face the much harder problem of breaking the encryption anyways.

Outlook and future work

As mentioned before, the extension of the provided tool for RAID-3, RAID-4 and RAID-6 systems is a viable next goal in the development along with nested RAID types (Wu and Chin, 2008).

In order to solve the hard disk order problem for RAID-0, some amount of hard disks must be permuted to generate potential disk orders from which images can be generated. In order to handle the factorial amount of work in reasonable time, our tool could be easily extended to distribute the workload in a cluster.

Finally, we could try to further improve the parameter detection by adding other block measures, such as the hash used by Garfinkel et al. (2010) as mentioned in Related work.

References

Arpaci-Dusseau RH, Arpaci-Dusseau AC. Operating systems: three easy pieces. Popular Books Publishing; 2012.

M. Belhadj, R. D. Daniels, D. K. Umberger, Raid rebuild using most vulnerable data redundancy scheme first, US Patent 6,516,425 (Feb. 4 2003).

Chen PM, Lee EK, Gibson GA, Katz RH, Patterson DA. Raid: high-performance, reliable secondary storage. ACM Comput Surv 1994; 26:145–85.

Corbett P, English B, Goel A, Grcanac T, Kleiman S, Leong J, et al. Row-diagonal parity for double disk failure correction. In: Proceedings of the 3rd USENIX conference on file and storage technologies; 2004. p. 1–14.

Devroye L. A probabilistic theory of pattern recognition31. Springer Science & Business Media; 1996.

Garfinkel S, Nelson A, White D, Roussev V. Using purpose-built functions and block hashes to enable small block and sub-file forensics. Digit Investig 2010;7:S13–23.

Goel A, Corbett P. Raid triple parity. SIGOPS Oper Syst Rev 2012;46(3): 41–9. http://dx.doi.org/10.1145/2421648.2421655. http://doi.acm.org/10.1145/2421648.2421655.

N. Hart, Method, disaster recovery record, back-up apparatus and raid array controller for use in restoring a configuration of a raid device, US Patent App. 10/152,340 (May 22 2002).

Katz RH. Raid: a personal recollection of how storage became a system. Ann Hist Comput IEEE 2010;32(4):82–7.

O. Kiselev, J. A. Colgrove, Automated recovery from data corruption of data volumes in parity raid storage systems, US Patent 7,146,461 (Dec. 5 2006).

Lee EK. Software and performance issues in the implementation of a raid prototype. Tech Rep DTIC Doc 1990.

Patterson DA, Gibson G, Katz RH. A case for redundant arrays of inexpensive disks (raid). SIGMOD Rec 1988;17(3):109–16. http://dx.doi.org/10.1145/971701.50214. http://doi.acm.org/10.1145/971701.50214.

Perklin M. Anti-forensics and anti-anti-forensics. https://www.defcon.org/images/defcon-20/dc-20-presentations/Perklin/DEFCON-20-Perklin\-AntiForensics.pdf; 2012. visited2015-10-05.

Salomon D, Motta G. Handbook of data compression. Springer Science & Business Media; 2010.

Shannon CE. A mathematical theory of communication. ACM Sigmob Mob Comput Commun Rev 2001;5(1):3–55.

G. Wu, R. Chin, Non-volatile memory storage system, US Patent App. 12/271,885 (Nov. 15 2008).

Xiang L, Xu Y, Lui JC, Chang Q. Optimal recovery of single disk failure in RDP code storage systems. SIGMETRICS Perform Eval Rev 2010;38(1): 119–30. http://dx.doi.org/10.1145/1811099.1811054. http://doi.acm.org/10.1145/1811099.1811054.

Xiang L, Xu Y, Lui JCS, Chang Q, Pan Y, Li R. A hybrid approach to failed disk recovery using raid-6 codes: algorithms and performance evaluation. Trans Storage 2011;7(3):11:1–11:34. http://dx.doi.org/10.1145/2027066.2027071. http://doi.acm.org/10.1145/2027066.2027071.

Xiong J, Xiong J, Claramunt C. A spatial entropy-based approach to improve mobile risk-based authentication. In: Proceedings of the 1st ACM SIGSPATIAL international workshop on privacy in geographic

information collection and analysis; 2014. 3:1–3:9. http://dx.doi.org/10.1145/2675682.2676400. GeoPrivacy '14, ACM, New York, NY, USA, http://doi.acm.org/10.1145/2675682.2676400.

Yokota T, Ootsu K, Baba T. Introducing entropies for representing program behavior and branch predictor performance. In: Proceedings of the 2007 workshop on experimental computer science; 2007. http://dx.doi.org/10.1145/1281700.1281717. ExpCS '07, ACM, New York, NY, USA, http://doi.acm.org/10.1145/1281700.1281717.